Gwen Salaün
Bernhard Schätz (Eds.)

Formal Methods for Industrial Critical Systems

LNCS 6959

16th International Workshop, FMICS 2011
Trento, Italy, August 2011
Proceedings

Springer

Lecture Notes in Computer Science 6959

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Gwen Salaün Bernhard Schätz (Eds.)

Formal Methods for Industrial Critical Systems

16th International Workshop, FMICS 2011
Trento, Italy, August 29-30, 2011
Proceedings

Springer

Volume Editors

Gwen Salaün
Grenoble INP - INRIA - LIG
Montbonnot Saint-Martin, France
E-mail: gwen.salaun@inria.fr

Bernhard Schätz
fortiss GmbH
München, Germany
E-mail: schaetz@fortiss.org

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24430-8 e-ISBN 978-3-642-24431-5
DOI 10.1007/978-3-642-24431-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011936880

CR Subject Classification (1998): D.2.4, D.2, D.3, C.3, C.2.4, F.3, I.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at FMICS 2011, the 16th International Workshop on Formal Methods for Industrial Critical Systems, taking place August 29–30, 2011, in Trento, Italy. Previous workshops of the ERCIM Working Group on Formal Methods for Industrial Critical Systems were held in Oxford (March 1996), Cesena (July 1997), Amsterdam (May 1998), Trento (July 1999), Berlin (April 2000), Paris (July 2001), Malaga (July 2002), Trondheim (June 2003), Linz (September 2004), Lisbon (September 2005), Bonn (August 2006), Berlin (July 2007), L'Aquila (September 2008), Eindhoven (November 2009), and Antwerp (September 2010). The FMICS 2011 workshop was co-located with the 19th IEEE International Requirements Engineering Conference (RE 2011).

The aim of the FMICS workshop series is to provide a forum for researchers who are interested in the development and application of formal methods in industry. In particular, these workshops bring together scientists and engineers who are active in the area of formal methods and are interested in exchanging their experiences in the industrial usage of these methods. These workshops also strive to promote research and development for the improvement of formal methods and tools for industrial applications.

Thus, topics of interest for FMICS 2011 include, but are not limited to:

- Design, specification, code generation and testing based on formal methods
- Methods, techniques and tools to support automated analysis, certification, debugging, learning, optimization and transformation of complex, distributed, real-time systems and embedded systems
- Verification and validation methods that address shortcomings of existing methods with respect to their industrial applicability (e.g., scalability and usability issues)
- Tools for the development of formal design descriptions
- Case studies and experience reports on industrial applications of formal methods, focusing on lessons learned or identification of new research directions
- Impact of the adoption of formal methods on the development process and associated costs
- Application of formal methods in standardization and industrial forums

This year, we received 39 submissions. Papers underwent a rigorous review process, and received three or four review reports. After the review process, the international Program Committee of FMICS 2011 decided to select 16 papers for presentation during the workshop and inclusion in these proceedings. The workshop featured two invited talks by Leonardo de Moura (Microsoft Research, USA) and Joost-Pieter Katoen (RWTH Aachen University, Germany); this volume includes two extended abstracts written by our invited speakers.

Following a tradition established over the past few years, the European Association of Software Science and Technology (EASST) offered an award to the best FMICS paper. This year, the reviewers selected the contribution by Thomas Reinbacher, Joerg Brauer, Martin Horauer, Andreas Steininger and Stefan Kowalewski on "Past Time LTL Runtime Verification for Microcontroller Binary Code." Further information about the FMICS working group and the next FMICS workshop can be found at: http://www.inrialpes.fr/vasy/fmics.

We would like to thank the local organizers Anna Perini and Angelo Susi (Fondazione Bruno Kessler - IRST, Trento, Italy) for taking care of all the local arrangements to host FMICS in Trento, the ERCIM FMICS working group Coordinator Alessandro Fantechi (Univ. degli Studi di Firenze and ISTI-CNR, Italy) for guiding us when necessary, Jan Olaf Blech (fortiss GmbH, Germany) for acting as Publicity Chair and coordinating the publication process, EasyChair for supporting the review process, Springer for taking over the publication, all the members of the Program Committee for their great work during the review process, the external reviewers for their participation during the review process of the submissions, all the authors for submitting papers to the workshop, and the authors who participate in the workshop in Trento. All these people contributed to the success of the 2011 edition of FMICS.

August 2011

Bernhard Schätz
Gwen Salaün

Organization

Program Committee

María Alpuente	UPV, Spain
Jiri Barnat	Masaryk University, Czech Republic
Josh Berdine	Microsoft Research, Cambridge, UK
Jan Olaf Blech	fortiss GmbH, Germany
Rance Cleaveland	University of Maryland, USA
Cindy Eisner	IBM Haifa Research Laboratory, Israel
Wan Fokkink	Vrije Universiteit Amsterdam, The Netherlands
Stefania Gnesi	ISTI-CNR, Italy
Holger Hermanns	Saarland University, Germany
Daniel Kaestner	AbsInt GmbH, Germany
Stefan Kowalewski	RWTH Aachen University, Germany
Daniel Kroening	Computing Laboratory, Oxford University, UK
Frederic Lang	INRIA Rhône-Alpes / VASY, France
Kim G. Larsen	Aalborg University, Denmark
Diego Latella	ISTI-CNR, Pisa, Italy
Timo Latvala	Space Systems, Finland
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Charles Pecheur	UC Louvain, France
Ernesto Pimentel	University of Malaga, Spain
Marco Roveri	FBK-irst, Italy
John Rushby	SRI International, USA
Gwen Salaün	Grenoble INP - INRIA - LIG, France
Thomas Santen	European Microsoft Innovation Center, Aachen, Germany
Bernhard Schätz	fortiss GmbH, Germany
Marjan Sirjani	Reykjavik University, Iceland
Jaco Van De Pol	University of Twente, The Netherlands
Helmut Veith	TU Wien, Austria

Additional Reviewers

Biallas, Sebastian	Crouzen, Pepijn
Blom, Stefan	Dražan, Sven
Bortolussi, Luca	Dräger, Klaus
Bozzano, Marco	Eisentraut, Christian
Brauer, Jörg	Escobar, Santiago
Bulychev, Peter	Fantechi, Alessandro
Ceska, Milan	Ferrari, Alessio

Haller, Leopold
Hartmanns, Arnd
Hatefiardakani, Hassan
He, Nannan
Heckmann, Reinhold
Ilic, Dubravka
Kant, Gijs
Ketema, Jeroen
Khakpour, Narges
Khosravi, Ramtin
Konnov, Igor
Loreti, Michele
Mateescu, Radu
Mazzanti, Franco
Mousavi, Mohammadreza
Mover, Sergio
Nimal, Vincent
Olsen, Petur
Ouederni, Meriem
Panizo, Laura
Pfaller, Christian
Poll, Erik
Reinbacher, Thomas
Romero, Daniel
Sabouri, Hamideh
Sanan, David
Schuppan, Viktor
Serwe, Wendelin
Steiner, Wilfried
Tautschnig, Michael
Ter Beek, Maurice H.
Timmer, Mark
Titolo, Laura
Tonetta, Stefano
Trachtenherz, David
Tumova, Jana
Varpaaniemi, Kimmo
Villanueva, Alicia
Voss, Sebastian
Zuleger, Florian

Table of Contents

Towards Trustworthy Aerospace Systems: An Experience Report 1
Joost-Pieter Katoen

Satisfiability at Microsoft 5
Leonardo de Moura

Lightweight Verification of a Multi-Task Threaded Server: A Case Study with the Plural Tool 6
Néstor Cataño and Ijaz Ahmed

Runtime Verification of Typical Requirements for a Space Critical SoC Platform 21
Luca Ferro, Laurence Pierre, Zeineb Bel Hadj Amor, Jérôme Lachaize, and Vincent Lefftz

Past Time LTL Runtime Verification for Microcontroller Binary Code 37
Thomas Reinbacher, Jörg Brauer, Martin Horauer, Andreas Steininger, and Stefan Kowalewski

A SAT-Based Approach for the Construction of Reusable Control System Components 52
Daniel Côté, Benoît Fraikin, Marc Frappier, and Richard St-Denis

Formal Safety Analysis in Industrial Practice 68
Ilyas Daskaya, Michaela Huhn, and Stefan Milius

Structural Test Coverage Criteria for Integration Testing of LUSTRE/SCADE Programs 85
Virginia Papailiopoulou, Ajitha Rajan, and Ioannis Parissis

Formal Analysis of a Triplex Sensor Voter in an Industrial Context 102
Michael Dierkes

Experiences with Formal Engineering: Model-Based Specification, Implementation and Testing of a Software Bus at Neopost. 117
Marten Sijtema, Mariëlle I.A. Stoelinga, Axel Belinfante, and Lawrence Marinelli

Symbolic Power Analysis of Cell Libraries 134
Matthias Raffelsieper and MohammadReza Mousavi

An Automated Semantic-Based Approach for Creating Tasks from Matlab Simulink Models 149
Matthias Büker, Werner Damm, Günter Ehmen, and Ingo Stierand

Performability Measure Specification: Combining CSRL and MSL 165
Alessandro Aldini, Marco Bernardo, and Jeremy Sproston

Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit Using CADP 180
Etienne Lantreibecq and Wendelin Serwe

Transforming SOS Specifications to Linear Processes 196
Frank P.M. Stappers, Michel A. Reniers, and Sven Weber

Formal Verification of Real-Time Data Processing of the LHC Beam Loss Monitoring System: A Case Study 212
Naghmeh Ghafari, Ramana Kumar, Jeff Joyce, Bernd Dehning, and Christos Zamantzas

Hierarchical Modeling and Formal Verification. An Industrial Case Study Using Reo and Vereofy 228
Joachim Klein, Sascha Klüppelholz, Andries Stam, and Christel Baier

Modeling and Verifying Timed Compensable Workflows and an Application to Health Care 244
Ahmed Shah Mashiyat, Fazle Rabbi, and Wendy MacCaull

Author Index 261

Towards Trustworthy Aerospace Systems: An Experience Report

Joost-Pieter Katoen

RWTH Aachen University, Software Modeling and Verification Group, Germany

1 Introduction

Building modern aerospace systems is highly demanding. They should be extremely dependable. They must offer service without interruption (i.e., without failure) for a very long time — typically years or decades. Whereas "five nines" dependability, i.e., a 99.999 % availability, is satisfactory for most safety-critical systems, for on-board systems it is not. Faults are costly and may severly damage reputations. Dramatic examples are known. Fatal defects in the control software of the Ariane-5 rocket and the Mars Pathfinder have led to headlines in newspapers all over the world. Rigorous design support and analysis techniques are called for. Bugs must be found as early as possible in the design process while performance and reliability guarantees need to be checked whenever possible. The effect of fault diagnosis, isolation and recovery must be quantifiable.

Tailored effective techniques exist for specific system-level aspects. Peer reviewing and extensive testing find most of the software bugs, performance is checked using queueing networks or simulation, and hardware safety levels are analysed using an profiled Failure Modes and Effects Analysis (FMEA) approach. Fine. But how is the consistency between the analysis results ensured? What is the relevance of a zero- bug confirmation if its analysis is based on a system view that ignores critical performance bottlenecks? There is a clear need for an integrated, coherent approach! This is easier said than done: the inherent heterogeneous character of on-board systems involving software, sensors, actuators, hydraulics, electrical components, etc., each with its own specific development approach, severely complicates this.

2 Modeling Using an AADL Dialect

About three years ago we took up this grand challenge. Within the ESA-funded COMPASS (COrrectness, Modeling and Performance of Aerospace SyStems) project, an overarching model-based approach has been developed. The key is to model on-board systems at an adequate level of abstraction using a general-purpose modeling and specification formalism based on AADL (Architecture Analysis & Design Language) as standardised by SAE International. This enables engineers to use an industry-standard, textual and graphical notation with precise semantics to model system designs, including both hardware as well as software components. Ambiguities about the meaning of designs are abandoned. System aspects that can be modeled are, amongst others,

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 1–4, 2011.

- (timed) hardware operations, specified on the level of processors, buses, etc.,
- software operations, supporting concepts such as processes and threads,
- hybrid aspects, i.e., continuous, real-valued variables with (linear) time-dependent dynamics, and
- faults with probabilistic failure rates and their propagation between components.

A complete system specification describes three parts: (1) nominal behavior, (2) error behavior, and (3) a fault injection—how does the error behavior influence the system's nominal behavior? Systems are described in a component-based manner such that the structure of system models strongly resembles the real system's structure. A detailed description of the language and its formal semantics can be found in [2].

3 Formal Verification

This coherent and multi-disciplinary modeling approach is complemented by a rich palette of analysis techniques. The richness of the AADL dialect gives the power to specify and generate a single system model that can be analysed for multiple qualities: reliability, availability, safety, performance, and their mixture. All analysis outcomes are related to the same system's perspective, thus ensuring compatibility. First and foremost, mathematical techniques are used to enable an early integration of bug hunting in the design process. This reduces the time that is typically spent on a posteriori testing —in on-board systems, more time and effort is spent on verification than on construction!— and allows for early adaptations of the design. The true power of the applied techniques is their almost full automation: once a model and a property (e.g., can a system ever reach a state in which the system cannot progress?) are given, running the analysis is push-button technology. In case the property is violated, diagnostic feedback is provided in terms of a counterexample which is helpful to find the cause of the property refutation. These model-checking techniques [1] are based on a full state space exploration, and detect all kinds of bugs, in particular also those that are due to the intricacies of concurrency: multiple threads acting on shared data structures. This type of bugs are becoming increasingly frequent, as multi-threading grows at a staggering rate.

4 Requirements

Whereas academic tools rely on properties defined in mathematical logic, a language that is major obstacle for usage by design engineers, COMPASS uses specification patterns [5]. These patterns act as parametrised "templates" to the engineers and thus offer a comprehensible and easy-to-use framework for requirement specification. In order to ensure the quality of requirements, they can be validated independently of the system model. This includes property consistency (i.e., checking that requirements do not exclude each other), and property assertion (i.e., checking whether an assertion is a logical consequence of the requirements).

5 Safety

Analysing system safety and dependability is supported by key techniques such as (dynamic) fault tree analysis (FTA), (dynamic) Failure Modes and Effects Analysis (FMEA), fault tolerance evaluation, and criticality analysis [4]. System models can include a formal description of both the fault detection and isolation subsystems, and the recovery actions to be taken. Based on these models, tool facilities are provided to analyze the operational effectiveness of the FDIR (Fualt Detection, Isolation and Recovery) measures, and to assess whether the observability of system parameters is sufficient to make failure situations diagnosable.

6 Toolset

All techniques and the full modeling approach are supported by the COMPASS toolset [3], developed in close cooperation with the Italian research institute Fondazione Bruno Kessler in Trento, and is freely downloadable for all ESA countries from the website `compass.informatik.rwth-aachen.de`. The tool is graphical, runs under Linux, and has an easy-to-use GUI.

7 Industrial Evaluation

The COMPASS approach and toolset was intensively tested on serious industrial cases by Thales Alenia Space in Cannes (France). These cases include thermal regulation in satellites and satellite mode management with its associated FDIR strategy. It was concluded that the modeling approach based on AADL provides sufficient expressiveness to model all hardware and software subsystems in satellite avionics. The hierarchical structure of specifications and the component-based paradigm enables the reuse of models. Also incremental modeling is very well supported. The RAMS analyses as provided by the toolset were found to be mature enough to be adopted by industry, and the corresponding results allowed the evaluation of design alternatives [6]. Current investigations indicate that the integrated COMPASS approach significantly reduces the time and cost for safety analysis compared to traditional on-board design processes.

Acknowledgement. We thank all co-workers in the COMPASS project for their contributions, in particular Thomas Noll and Viet Yen Nguyen (RWTH Aachen University), Marco Bozzano, Alessandro Cimatti and Marco Roveri (FBK, Trento), Xavier Olivé (Thales) and Yuri Yushstein (ESA). This research is funded by the European Space Agency via several grants.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability, and performance analysis of extended AADL models. The Computer Journal (March 2010), doi:10.1093/com

3. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M., Wimmer, R.: A model checker for AADL (tool presentation). In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 562–565. Springer, Heidelberg (2010)
4. Bozzano, M., Villafiorita, A.: Design and Safety Assessment of Critical Systems. CRC Press, Boca Raton (2010)
5. Grunske, L.: Specification patterns for probabilistic quality properties. In: Int. Conf. on Software Engineering (ICSE), pp. 31–40. ACM, New York (2008)
6. Yushstein, Y., Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Olivé, X., Roveri, M.: System-software co-engineering: Dependability and safety perspective. In: 4th IEEE International Conference on Space Mission Challenges in Information Technology (SMC-IT). IEEE CS Press, Los Alamitos (2011)

Satisfiability at Microsoft

Leonardo de Moura

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
leonardo@microsoft.com

Abstract. Constraint satisfaction problems arise in many diverse areas including software and hardware verification, type inference, static program analysis, test-case generation, scheduling, planning and graph problems. These areas share a common trait, they include a core component using logical formulas for describing states and transformations between them. The most well-known constraint satisfaction problem is *propositional satisfiability*, SAT, where the goal is to decide whether a formula over Boolean variables, formed using logical connectives can be made *true* by choosing *true*/*false* values for its variables. Some problems are more naturally described using richer languages, such as arithmetic. A supporting *theory* (of arithmetic) is then required to capture the meaning of these formulas. Solvers for such formulations are commonly called *Satisfiability Modulo Theories* (SMT) solvers.

Modern software analysis and model-based tools are increasingly complex and multi-faceted software systems. However, at their core is invariably a component using logical formulas for describing states and transformations between system states. In a nutshell, symbolic logic is *the calculus* of computation. The state-of-the art SMT solver, Z3, developed at Microsoft Research, can be used to check the satisfiability of logical formulas over one or more theories. SMT solvers offer a compelling match for software tools, since several common software constructs map directly into supported theories.

SMT solvers have been the focus of increased recent attention thanks to technological advances and an increasing number of applications. The Z3 solver from Microsoft Research is particularly prolific both concerning applications and technological advances. We describe several of the applications of Z3 within Microsoft, some are included as critical components in tools shipped with Windows 7, others are used internally and yet more are available for academic research. Z3 ranks as the premier SMT solver available today.

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, p. 5, 2011.

Lightweight Verification of a Multi-Task Threaded Server: A Case Study With The Plural Tool*

Néstor Cataño and Ijaz Ahmed

Carnegie Mellon University - Portugal, Madeira ITI
Campus da Penteada, Funchal, Portugal
{nestor.catano,ijaz.ahmed}@m-iti.org

Abstract. In this case study, we used the Plural tool to verify the design of a commercial multi-task threaded application (MTTS) implemented by Novabase, which has been used for massively parallelising computational tasks. The effort undertaken in this case study has revealed several issues related with the design of the MTTS, with programming practices used in its implementation, and with domain specific properties of the MTTS. This case study has also provided insight on how the analysis done by the Plural tool can be improved. The Plural tool performs lightweight verification of Java programs. Plural specification language combines typestates and access permissions, backed by Linear Logic. The Plural specifications we wrote for the MTTS are based on its code, its informal documentation, sometimes embedded in the code, and our discussions with Novabase's engineers, who validated our understanding of the MTTS application.

Keywords: Concurrency, Formal Methods, Parallelism, The Plural Tool, Verification.

1 Introduction

Hardware engineers and chip manufacturers are currently developing even bigger multi-core processors to develop desktops that will be massively parallel within the next few years. To take advantage of this new parallel technology, computer scientists are working on the development of programming languages and programming paradigms that exploit the massively parallel power provided by the (future) hardware. As an example of this, the Æminium research project is working in the development of a platform [16] that allows programmers to write concurrent-by-default programs and that supports massive parallelism. Æminium platform is under development and its evolution has been greatly influenced by the Plural tool [14], its predecessor. Plural provides support to *typestates* and *access permissions*. Typestates define protocols on finite state

* This work has been supported by the Portuguese Research Agency FCT through the CMU-Portugal program, R&D Project Æminium, CMU-PT/SE/0038/2008.

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 6–20, 2011.

machines [17]. Access permissions are abstractions describing how objects are accessed. For instance, a **Unique** access permission describes the case when a sole reference to a particular object exists, and a **Shared** access permission models the case when an object is accessed by multiple references. Access permissions are inspired by Girard's Linear Logic [11], hence, they can be used as part of specifications, and can be produced and consumed.

For this case study, we used the Plural tool for the specification and verification of a multi-task threaded server (MTTS), implemented by Novabase [13], which has extensively used for massively parallelising the processing of computational tasks. The MTTS has historically been a robust and reliable core application. The MTTS is part of several software applications used by Novabase's clientele, e.g. it is used in the financial sector to parallelise the archiving and processing of documents. MTTS has been implemented in Java. It utilises queues to store tasks, which are executed by a pool of threads.

Our goals for the case study included verifying the design of a massively parallel application, determining how well the Plural tool works on a complex commercial application - what kinds of specifications and code can and cannot be analysed by Plural. This case study further allowed us to make a list of desirable properties and features the Plural tool (and Æminium, its successor) must implement to become a more powerful and usable lightweight verification tool. The process of verifying MTTS with Plural revealed a series of issues related with good programming practises and with design decisions made in the implementation of the MTTS. These issues would not have been revealed otherwise, e.g. through direct code inspection. Furthermore, the specification we wrote can be used by Novabase to generate a collection of documents describing the behaviour of the MTTS, and the use of an automated tool like Plural increases our confidence on the correctness of the specifications. This documentation can be used to resolve differences between members of the quality assurance team of Novabase, their programmers, or any of those and Novabase' clientele itself, regarding the expected behaviour of the MTTS. The Plural specifications we wrote for the MTTS are based on our understanding of the MTTS application, built from direct inspection of its code and from our discussions with Novabase's engineers, who validated our understanding.

The rest of this paper is organised as follows. In the following, we present some related work. Section 2 introduces the specification language used by Plural (access permissions and typestates), and describes how verification is performed with the Plural tool. Section 3 presents the structure of the MTTS application. Section 4 presents the specification of the MTTS application and shows miscellaneous aspects of the verification of the MTTS with Plural. Section 4 includes a discussion on the limitations of the analysis performed by Plural and how we overcame these limitations. This discussion includes a list of desirable features regarding the analysis done by Plural. Section 5 discusses our current work on the implementation of some of these features.

Related Work. In previous work [8], we used JML to specify an electronic purse application written in the Java Card dialect of Java. JML is a behavioural interface specification language for Java [12]. Typestates can be regarded as JML abstract variables and, therefore, JML tools can be used to simulate the typestate verification of specifications. However, JML does not provide support for the reasoning about access permissions and JML's support for concurrency is rather limited. The work presented here is more complex than the work in [8], as it involves reasoning on concurrency properties of a system.

The Plural group has conducted several case studies on the use of typestates to verify Java I/O stream libraries [3] and Java database libraries [4]. The case studies show that Plural can effectively be used to check violation of APIs protocols. The case study presented in this paper takes a further step in considering a large commercial application with about fifty Java classes.

In [10], Robert DeLine and Manuel Fähndrich use the Fugue protocol checker on a relatively large .Net web based application. Likewise Plural, Fugue provides support to typestate verification, however, it does not provide support to access permissions. In [9], the Vault programming language is used to describe resource management protocols. Protocols can specify that certain operations must be performed in a certain order and that certain operations must be performed before accessing a given data object. The technique has been used on the interface between the Windows kernel and its device drivers.

2 Preliminaries

2.1 Plural Specification Language

Plural specification language combines typestates and access permissions specifications. Typestates define protocols on finite state machines [17]. Access permissions are abstract definitions on the capability of a method to access a particular state [3,5]. Plural uses access permissions to keep track of the various references to a particular object, and to check the types of accesses these references have. Accesses can be reading and writing (modifying). Plural provides support to five types of access permissions, namely, **Unique**, **Share**, **Immutable**, **Full**, and **Pure**. Figure 1 presents a taxonomy of how different access permissions can coexist, e.g. **Full** access to a referenced object allows the existence of any other reference with **Pure** access to the same referenced object.

- **Unique(x)**. It guarantees that reference **x** is the sole reference to the referenced object. No other reference exists, so **x** has exclusive reading and modifying (writing) access to the object.
- **Full(x)**. It provides reference **x** with reading and modifying access to the referenced object. Additionally, it allows other references to the object (called aliases) to exist and to read from it, but not to modify it.
- **Share(x)**. Its definition is similar to the definition of **Full(x)**, except that other references to the object can further modify it.

This reference	Other references
Unique	∅
Full	**Pure**
Share	**Share**, **Pure**
Pure	**Full**, **Share**, **Pure**, **Immutable**
Immutable	**Pure**, **Immutable**

Current permission		Access through
read/write	read-only	other permission
Unique	-	none
Full	**Immutable**	read-only
Share	**Pure**	read/write

Fig. 1. Simultaneous access permissions taxonomy [3]

- **Pure(x)**. It provides reference **x** with reading-only access to the referenced object. It further allows the existence of other references to the same object with read-only access or read-and-modify access.
- **Immutable(x)**. It provides **x** and any other existing reference to the same referenced object with non-modifying access (read-only) to the referenced object. An **Immutable** permission guarantees that all other existing references to the referenced object are also immutable permissions.

Access permissions are inspired by Girard's Linear Logic [11], hence, they can be used, produced and consumed. In Plural, typestates are declared with the aid of the **@ClassStates** clause. Method specifications are written with the aid of the **@Perm** clause, composed of a "requires" part, describing the resources required by a method to be executed, and an "ensures" part, describing the resources generated after method execution. So, the Linear Logic formula $P \multimap Q$ is written as **@Perm**(requires="**P**", ensures="**Q**"). The semantics of the operator $\otimes$ of Linear Logic, which denotes simultaneous occurrence of resources, is captured by the operator "*". **P** and **Q** above are specifications such as **Unique**(x) **in A** * **Full**(y) **in B**, which requires (ensures) that reference "x" has **Unique** permission to its referenced object, which should be in state **A**, and simultaneously requires (ensures) that "y" has **Full** permission to its referenced object, which should be in state **B**. The semantics of the additive conjunction operator "&" of Linear Logic, which represents the alternate occurrence of resources, is captured by the use of a **@Cases** specification, the decision of which is made according to a required resource in one of its **@Perm** specifications. The additive disjunction operator $\oplus$ of Linear Logic is modelled by the use of a **@Cases** specification, the decision of which is made according to an ensured resource by one of its **@Perm**.

Figure 2 illustrates an example of specification with Plural taken from the case study. Class Task models a generic processing task in the MTTS. The internal information about the task is stored in an object "data" of type MttsTaskDataX. We identify four possible typestates a task can be, namely, **Created**, **Ready**,

```
@ClassStates({
 @State(name = ``Created'', inv = ``data == null''),
 @State(name = ``Ready'', inv = ``data != null''),
 @State(name = ``Running'', inv = ``data != null''),
 @State(name = ``Finished'', inv = ``data == null'')
})
public Task implements AbstractTask {
  private MttsTaskDataX data;

  @Perm(ensures = ``Unique(this) in Created'')
  public Task() { ... }

  @Perm(requires = ``Full(this) in Created * #0 != null'',
        ensures = ``Full(this) in Ready'')
  public void setData(MttsTaskDataX data) { ... }

  @Perm(requires = ``Full(this) in Ready'',
        ensures = ``Full(this) in Finished'')
  public void execute() throws Exception { ... }
}
```

Fig. 2. Example of a specification for a generic task

Running, and **Finished**. The state **Created** is the initial state of any task. This uniqueness property is enforced by the class constructor. A task is in state **Ready** once it has been given some data to be run. It is in state **Running** when it is running, and it is in state **Finished** when it has been executed and the task data has been consumed. The constructor of class Task creates a **Unique** object that is initially in state **Created**. Method "setData()" requires **this** to have **Full** permission on its referenced object, which should be in state **Created**, and simultaneously requires that its first parameter is different than **null**. The operator "*****" combines several specifications. Expression "#i" stands for the parameter number "i+1" of a method. Method "execute()" requires **this** to have **Full** permission and to be in state **Ready**, and ensures that **this** will have **Full** permission on its referenced object, which will be in state **Finished**. Additionally (not shown in this example), in Plural, the clause **@Cases** allows the annotation of several **@Perm** specifications for a method.

2.2 Checking Programs with Plural

Plural is a typestate specification and checker tool, implemented as a plug-in of Eclipse [14]. The Plural tool takes a Java program annotated with Plural specifications and checks whether the program complies with its specifications. Plural performs several types of program analysis, e.g. fractional analysis (influenced by Boyland's work in [6]), hence, access permissions can be split in several more *relaxed* permissions and then joined back again to form more *restrictive* permissions. Plural has also a simple effects-analyser that checks if a particular method has side-effects, and an annotation analysis tool that checks whether annotations are well-formed.

Plural employs a *packing* and *unpacking* object mechanism that is used to transition objects into valid states in which invariants can be checked. Plural implementation of the packing and unpacking mechanism has been influenced

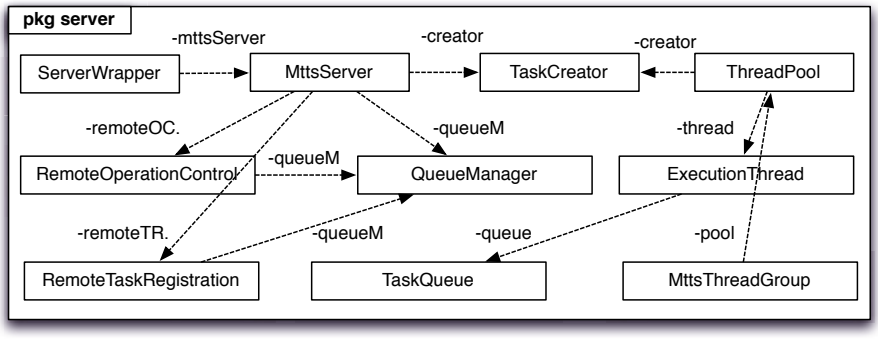


Fig. 3. The server package

by the work of M. Barnett and al. in [2]. Hence, Plural packs receiver objects to some state before methods are called. This ensures that objects are seen in consistent states during method calls.

3 General Outline of the MTTS Application

The MTTS is the core of a task distribution server that is used to run tasks over different execution threads. The core is used in the financial sector to process bank checks in parallel with time bound limits. MTTS' implementation is general in the sense that it makes no assumptions on the nature of the running tasks. The MTTS organises tasks through queues and schedules threads to execute the task queues. Tasks are stored in databases.

The MTTS is a typical client server application, which is divided into 3 main components, namely, TaskRegistration, RemoteOperationControl and QueueManager. MTTS' clients use the TaskRegistration component to register tasks. This component stores the registered tasks in a database. The QueueManager component implements some working threads that fetch and execute tasks. The RemoteOperationControl component is used to monitor and to control the progress of the tasks. Every queue implements a mutex manager algorithm to synchronise tasks.

Implementation of the MTTS. The MTTS is composed of three main packages, namely, mtts-api, il and server. The structure of the server Package is shown in Figure 3. The mtts-api package models tasks and queues. Class Task implements tasks and QueueInfo implements queues. Class IMutexImp in the il (intelligent lock) package implements a mutex algorithm to synchronise tasks, and class MutexManager creates and destroys locks. Lock status and statistics are implemented in classes IMutexStatus and IMutexStatistics respectively.

The server package is the main package of the MTTS application and uses features implemented by the other packages. The server package implements

code that fetches tasks from the database and distributes them through different threads. Class ServerWrapper runs the server as a system service and Class MttsServer implements the basic functionality to start and stop the server. Class TaskCreator and class QueueManager create tasks and manage queues respectively. Class RemoteTaskRegistration provides an interface to remotely register tasks and class RemoteOperationControl provides an interface to clients to remotely view the progress of tasks. Class ThreadPool keeps a list of class ExecutionThread objects that execute running threads. Class DBConnection implements the basic features to communicate with database.

4 Specification and Verification of MTTS

4.1 The General Specification Approach

The specification of the MTTS application is based on its informal documentation, sometimes embedded in the code as comments, and on our discussions with Novabase's engineers. After our discussions with Novabase's engineers took place, we wrote a technical report describing the architecture of the MTTS application [1]. The report was then validated by Novabase.

Since Plural performs a modular analysis of programs, we commenced writing specifications, starting from the most basic classes of the MTTS, e.g. classes that are inherited from or are used by other classes. Since the specification of more complex classes depends on the specification of the most basic ones, we provided basic classes with a sufficiently detailed specification. We specified the basic packages mtts-api and il first and specified package server last. Because Plural does not include a specification for Java standard classes, e.g. List and Map, we wrote specifications for these Java classes as well. We also wrote specifications for Java classes related with database interaction, e.g. Connection and DriverManager. In the following, we present and discuss some of the specifications of the three main packages of the MTTS and discuss miscellaneous aspects of the specification and verification of the MTTS application with Plural.

4.2 Miscellaneous Aspects of the Specification of the MTTS

Processing Tasks. Figure 2 presents an excerpt of the specification of class Task (see Section 2.1). There are some miscellaneous aspects about the specification of this class that are worthy to mention. Although we would like to distinguish typestates **Ready** and **Running**, their associated invariants are the same. In Plural, if an object is in state **Ready**, then "data" is different than **null**. However, the opposite direction is not necessarily **true**. If one wished to fully distinguish these two typestates then one could add conditions "isready" and "!isready" to their respective invariants. But then one would need to modify the source code of class Task by creating a boolean variable "ready" and to keep track of the value of this variable through the code of class Task. This is error-prone and we further wanted to keep the source code of the MTTS intact as much as possible.

The specification of class Task ensures that a task cannot be execute twice. Only the class constructor leaves a task in state **Created**. Only method "setData" transitions a task from **Created** to **Ready**. And a task needs to be in state **Ready** to be executed. The specification also ensures that "setData" must be called before "execute()".

Mutual Exclusion. Method "acquire()" acquires a lock and method "release()" gives the lock up. Method "acquire()" is the only class method that transitions into typestate **Acq**, and method "release()" is the only class method that takes an object from typestate **Acq** into typestate **FStat**. These methods are defined in class IMutex. Mutual exclusion to a critical section is ensured by enclosing the code of the critical section between a call to method "acquire()" and a call to method "release()". Hence, two different threads cannot execute a critical section simultaneously. If a first thread acquires a lock by successfully completing a call to "acquire()", a second thread can only acquire the lock after the first thread has released it.

```
@Full(requires=''FStat'', ensures=''Acq'')
public abstract void acquire() { }

@Full(requires=''Acq'', ensures=''FStat'')
public abstract void release() { }
```

An object of type ExecutionThread (see below) is in state **FullMutex** if it has **Full** access permission to field "mutex" of type IMutex. Hence, mutual exclusion to a certain code (e.g. method "doErrorRecovery") is attained by enclosing it between a call to "mutex.acquire()" and a call to "mutex.release()".

```
@ClassStates({
  @State(name=''FullMutex'', inv=''Full(mutex) in FStat''),
})
class ExecutionThread extends Thread {
 private IMutex mutex;

 @Perm(requires = ''FullMutex'', ensures = ''FullMutex'')
 private void doErrorRecovery(Exception e) {
   try { mutex.acquire(); ... }
   finally { ... mutex.release(); }
 }
}
```

Absence of Deadlocks. The Plural specification provided to methods "acquire()" and "release()" ensures that if a thread has acquired a lock, then the thread needs to release the lock before another thread can acquire the lock. This is a source for deadlocks: one needs to check that an acquired lock is eventually released. Plural does not provide support to reachability analysis so we did not prove the absence of deadlocks in general, but only in particular settings, e.g. by direct code inspection. As an example of this, the code of method "doErrorRecovery" in class "ExecutionThread" below is enclosed between a call to "acquire()"

and a call to "release()". This was often the case for methods in class "ExecutionThread". Method "doErrorRecovery" uses a Java try-catch-finally statement to ensure that the "release()" method is always finally called regardless of the method termination status (normal or exceptional). Plural analysers take the semantics of the try-catch-finally Java statement into account.

Destroying a Non-Released Lock. Method destroy(IMutex m) in class MutexManager removes a mutex from the list of mutexes. However, destroying (removing) a mutex can lead the system to a deadlock (or to a state that might enable some abnormal behaviour) as the thread that acquired the lock will never be able to release the lock and so threads waiting for the thread to release the lock will await forever. To ensure that a mutex is not destroyed before it is first released, we added the specification **Full**(#0) **in NotAcq** to the "requires" part of method "destroy" ("#0" refers to the first parameter of the method, i.e. m). Therefore, Plural will generate an error for any code that calls method "destroy" with a mutex object "m" in a state other than **NotAcq**.

Reentrant Mutexes. According to the implementation and the documentation of the MTTS, although two different threads cannot acquire the same lock, a single thread can acquire the same lock several times. Class IMutexImp implements interface IMutex. It declares a thread field "o" that keeps track of the thread that owns the lock, and an integer variable "nesting" that keeps track of the number of times the owner thread has acquired the lock. From the implementation of class IMutexImp, it appears evident that "nesting" is 0 whenever object "o" is **null** (a class invariant property). We define a typestate **NestAcq** (acquired several times by the same thread) related to the invariant "o != **null** * nesting $>$ 1" and use this typestate in the specification of all the methods of the class, e.g. "acquire()" and "release()". However, Plural does not provide support to integer arithmetic. We then thought of modifying the code of class IMutexImp to declare and use a boolean variable "nested" to be **true** whenever "nesting" is greater than 1, and modifying the invariant associated to the typestate **NestAcq** to be "o != **null** * nested==**true**". This approach however is error-prone: it requires us to set "nested" accordingly all through class IMutexImp whose code is large.

An additional problem related to the specification of reentrant mutexes has to do with the analysis performed by Plural. We describe this problem with the aid of the specification of the method "release()" below. This method requires the receiver object to be in state **NestAcq** that means"nested" !=**false**. This indicates that if-statement in method "release()" can never be executed, and hence the receiver object remains in the state **NestAcq**. However, the Plural tool issues a warning saying that it cannot establish the post-typestate specification. This proves another limitation of the analysis performed by Plural. To determine that the if-statement is never executed, it is necessary to analyse the invariant property associated with the definition of the **NestAcq** typestate. The actual

implementation of Plural does not perform such data-flow analysis. Due to all these limitations in the analysis performed by Plural, the actual specification for class IMutexImp does not introduce a **NestAcq** typestate.

```
@Full(requires=''NestAcq'', ensures =''NestAcq'')
public void release() {
  if(o != null && nested==false) { ... }
}
```

Plural and Good Programming Practices. Class ExecutionThread declares a boolean variable "terminate" that is used to determine whether the thread has finished its execution or not. The variable is not explicitly initialised in its declaration, yet according to the Java specification language its default value is **false**. Typestate **ThreadCreated** represents the state in which a thread has just been created. The constructor of class ExecutionThread does not set variable "terminate". Despite the fact that the initial value of variable "terminate" is **false**, Plural issues an error for the execution of the constructor of class ExecutionThread. This error states that the object cannot be packed to typestate **ThreadCreated**.

Although this shows a bug in the Plural tool, we report it as a programming bad practice. Programmers should explicitly initialise variables to their intended value, thus avoiding relying on the underlying compiler or on external tools, e.g. external typestate analysers like Plural. In this sense, Plural can be used to enforce initialisation of class variables.

```
@ClassStates({
 @State(name=''ThreadCreated'', inv=''terminate==false''),
  ...
})
class ExecutionThread extends Thread {
 private boolean terminate;
  ...
 @Perm (ensures = ''Unique(this) in ''ThreadCreated")
 ExecutionThread(...) { ... }
  ...
}
```

Specification of Standard Libraries. The MTTS stores tasks and related information into a database. The DBConnection class of MTTS implements the basic features that support communication with databases. Plural does not furnish specification of standard Java classes such as Connection and DriverManager, so we needed to write specifications for these classes as well. The specification of these classes allowed us to prove that the MTTS adheres to general protocols of database interaction. For instance, we proved that a connection is always open whenever database operations such as fetching a task from the database and updating task information stored in the database are taking place.

Abstract class MttsConnection below presents part of the specification we wrote for class MttsConnection. Class MttsConnection defines a root typestate **Connection** with two sub-typestates **OpenConnection** and **ClosedConnection**, modelling an open and closed database connection respectively. According

to the Java specification language, method "open()" can be called on an object that is in state **OpenConnection**, and "close()" can be called on an object that is in state **ClosedConnection**. The specification of other standard libraries was conducted in a similar way. They were all specified in Java abstract classes.

```
@Refine({
 @States(dim=``Connection'',
         value={``OpenConnection'', ``ClosedConnection''})
})
public abstract class MttsConnection {
  @Perm(ensures = ``Unique(this) in OpenConnection'')
  MttsConnection() { }

  @Full(value=``Connection'', ensures=``OpenConnection'')
  public abstract void open() throws java.sql.SQLException;

  @Full(value=``Connection'', ensures=``ClosedConnection'')
  public abstract void close() throws java.sql.SQLException;
}
```

Checking for Non-Nullness. One of the most common properties to verify is the one restricting values to be different than **null**. For instance, method "setName" (see below) in class "MttsTaskDataX" restricts parameter "name" to be different than **null**. This parameter is used to set the internal name of the underlying task. Plural issues an error for any call to method "setName" with a parameter that cannot be proved to be different than **null**.

```
@Perm(requires = ``#0 != null'')
public void setName(String name) { ... }
```

Starting and Shutting Down the MTTS Server. Class MttsServer is the main class of the MTTS application. It implements methods "start()" and "stop()" to start and to shutdown the server. It declares three variables "OpControlRemote", "TaskRegistrationRemote" and "queueManager" to manage the three major features of the server: control of remote operations, task registration, and the queue manager, respectively.

We wanted to check some design consistency aspects of the server, e.g. the server is in its starting state **ServerStart** if and only if its three components are in their starting states **TStart**, **CStart** and **QStart** respectively. Similarly, we wanted to prove that the MTTS server is in state **TShutdown** if and only its three components are in their respective shutdown states **TShutdown**, **CShutdown** and **QShutdown**. Although, the implementation of method "start()" verified the definition of typestate **ServerStart**, the implementation of method "stop()" did not verify the definition of typestate **ServerShutdown**. Method "stop()" does not shutdown all its components but only the "queueManager".

We report this as a flaw in the design of the MTTS server application. Due to the size of the MttsServer class and all the classes it uses, discovering this design flaw would not be possible through direct code inspection.

```
@ClassStates({
 @State(name=''ServerStart'',
        inv=''Full(queueManager) in QStart *
             Full(OpControlRemote) in CStart *
             Full(TaskRegistrationRemote) in TStart''),
 @State(name=''ServerShutdown'',
        inv=''Full(queueManager) in QShutdown
             Full(OpControlRemote) in CShutdown *
             Full(TaskRegistrationRemote) in TShutdown)''})
class MttsServer {
  private OpControl OpControlRemote;
  private TaskRegistration TaskRegistrationRemote;
  private QueueManager queueManager;

  @Perm(ensures=''Full(this) in ServerStart'')
  public void start() throws MttsException { ... }

  @Perm(requires=''Full(this) in ServerStart'',
        ensures=''Full(this) in ServerShutdown'')
  public void stop() throws MttsException {
   queueManager.shutdown();
  }
  ...
}
```

4.3 Discussion on the Limitations of Plural

We verified significant properties of the MTTS application ranging from simple non-null properties to absence of deadlocks and mutual exclusion to a critical section. Our experience dictates that Plural is a practical tool that can effectively be used to verify complex system properties that are harder to verify using other automated approaches. We also used Plural to check design decision in the implementation of the MTTS. We consider that the specification we have written can be used to enhance the quality of the implementation of the MTTS application. The mere exercise of writing abstractions (typestates) for an application forced us to fully understand and evaluate the MTTS application. The incompleteness of the written specifications are mainly due to the incompleteness of the analysis performed by Plural.

In the following, we summarise Plural limitations. Some of these limitations have already been discussed in previous sections.

- The Plural tool does not provide support to the analysis of programs with loops. For instance, method "run()" in class ExecutionThread loops while no termination request has not been placed - "while(!terminate){...}". To check method "run()", we modified it so that it loops once at the most - "if(!terminate){ ... }". The if-statement abstracts the while-loop statement. This abstraction is a source of incompleteness in the analysis we performed of the MTTS application. Nonetheless, abstracting loops as conditional statements is often a decision made in the implementation of formal methods tools, e.g. the ESC/Java tool [7] implements a similar approach to deal with loops.
- Plural does not provide support to integer arithmetic. So, one cannot define invariants that use integer variables. Plural provides support to the analysis

of boolean expressions that check equality or non-equality of references, or to boolean expressions that check (non-) nullness of references.
In the implementation of the MTTS, class IMutexImp implements a mutex algorithm that is used to synchronise threads. Mutexes can be acquired or released. Thus, if a thread acquires a lock then no other thread can acquire the same lock. A thread can acquire a lock several times. So, it must release the lock the same number of times it acquired it for any other thread to (eventually) be able to acquire the lock. However, in Plural it is not possible to define a typestate that describes the situation when a thread has acquired a lock several times as this will require the invariant related to the typestate to rely on an integer arithmetic expression "nesting > 1".

- Plural does not implement a strong specification typechecker, so programmers can unconsciously write specifications that include misspelled (nonexistent) typestates, and the Plural analysers can unconsciously use the misspelled typestate in their analysis. Plural does not issue any error on a specification that uses a nonexistent typestate.
- Plural does not provide support to reachability analysis. If a thread object is in state **Acq**, will the object ever be in state **NotAcq**? Plural does not provide support to reachability analysis. Nonetheless, in practice, for small classes, one can inspect the code and trace how states evolve. For large classes and large pieces of code this becomes impossible.
- Method "execute()" in Figure 2 transitions a task object from typestate **Ready** to typestate **Running**, and thereafter to typestate **Finished**. These two transitions occur both within method "execute()". Typestate **Ready** is required by method "execute()" and typestate **Finished** is produced by method "execute()". **Running** is an intermediate typestate for which the specification of method "execute()" does not provide any information. Not being able to reason about intermediate program states is a limitation of the analysis performed by the Plural tool. The information about intermediate states can be used by programmers (and tools) to assert certain facts that otherwise cannot be asserted. For instance, in the verification of the MTTS, we could not specify the property stating that a running task cannot be deleted.

5 Conclusion and Future Work

The MTTS is a relatively large size commercial application that implements a server with a thread pool that runs processing tasks. The specification and verification of the MTTS was a challenging and laborious task. The authors spent about six months writing the specifications and verifying the MTTS application with Plural. We specified and verified forty nine Java classes with 14451 lines of Java code and 546 lines of program specifications written in Plural. The automation of the analyses performed by the Plural tool ranges from a couple of milliseconds for the verification of small classes to a couple of minutes for the verification of large classes, e.g. the server class. The first author had previous

experience in the specification and verification of Java applications using JML but did not have any previous experience with Plural. The second author had no previous experience in the use of formal methods tools. The code of the MTTS was not originally documented so we needed to write its documentation prior to its specification and verification. We kept the code of the MTTS unchanged as much as possible and tried to keep the semantics of the code intact whenever we introduced any changes to it.

This is the first case study on the use of Plural for the verification of a commercial large sized application. Some of the limitations of Plural (see discussion in Section 4.3 for a full list) hampered our specification and verification work. Nonetheless, we managed to specify and verify important design properties backing the implementation of the MTTS application. The written typestate specifications can further be used to generate a collection of documents describing the behaviour of the MTTS, which can be used for the quality assurance team of Novabase for different purposes.

We are currently working on the implementation of some features to overcome some of these limitations. The two authors and Radu Siminiceanu, at the National Institute of Aerospace in Virginia, are currently working on a formal methods based approach to the verification of Plural and Æminium specifications. Hence, specifications are translated into an abstract state-machine that captures all possible behaviour of the specifications, and the EVMDD (Edge-Valued MDD) symbolic model-checker [15] is used to check the specifications. Furthermore, we are working on a Petri net inspired semantics to represent access permissions. The translation is carried out for specifications alone, regardless of the program source code. This ongoing work enables reachability analysis of Plural specifications and the checking for absence of misspelled specifications.

Writing program specifications for medium sized or large applications is a laborious and sometimes complex task. To help programmers write Plural specifications, we are currently working on a prototype tool that infers likely access permissions automatically. Roughly speaking, we analyse a Java program and store in a graph the information on how program objects are read or written. The graph is stored as an XML file, which is then used to generate the access permissions. We are currently working on the implementation of the prototype tool as an Eclipse plug-in.

Acknowledgements. We thank Jonathan Aldrich and Nels Beckman at Carnegie Mellon University, and Filipe Martins, Manuel Beja, and Paulo Casanova at Novabase for useful feedback on the work presented in this paper.

References

1. Ahmed, I., Cataño, N.: Architecture of Novabase' MTTS application. Technical report, The University of Madeira (2010), `http://www3.uma.pt/ncatano/aeminium/Documents_files/mtts.pdf`
2. Barnett, M., DeLine, R., Fhndrich, M., Rustan, K., Leino, M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3 (2004)

3. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA, pp. 301–320 (2007)
4. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API protocol checking with access permissions. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 195–219. Springer, Heidelberg (2009)
5. Bierhoff, N.B.K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. In: OOPSLA (2008)
6. Boyland, J.: Checking interference with fractional permissions. In: Proceedings of the 10th International Conference on Static Analysis, SAS, pp. 55–72 (2003)
7. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications (2003)
8. Cataño, N., Wahls, T.: Executing JML specifications of java card applications: A case study. In: 24th ACM Symposium on Applied Computing, Software Engineering Track (SAC-SE), Honolulu, Hawaii, March 8-12, pp. 404–408 (2009)
9. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI, pp. 59–69 (2001)
10. DeLine, R., Fähndrich, M.: The Fugue protocol checker: Is your software baroque (2003)
11. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
12. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT (Software Engineering Symposium) 31(3), 1–38 (2006)
13. Novabase, `http://www.novabase.pt`
14. The Plural Tool, `http://code.google.com/p/pluralism/`
15. Roux, P., Siminiceanu, R.: Model checking with edge-valued decision diagrams. In: NASA Formal Methods Symposium (NFM), NASA/CP-2010-216215, pp. 222–226. Langley Research Center, NASA (April 2010)
16. Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: Conference on Object-Oriented Programming Systems and Applications, OOPSLA, pp. 933–940 (2009)
17. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering 12, 157–171 (1986)

Runtime Verification of Typical Requirements for a Space Critical SoC Platform*

Luca Ferro[1], Laurence Pierre[1], Zeineb Bel Hadj Amor[1], Jérôme Lachaize[2], and Vincent Lefftz[2]

[1] TIMA (CNRS-INPG-UJF), 46 Av. Félix Viallet, 38031 Grenoble cedex, France
[2] EADS Astrium Satellites, Central Engineering, 31402 Toulouse cedex, France

Abstract. SystemC TLM (Transaction Level Modeling) enables the description of complex Systems on Chip (SoC) at a high level of abstraction. It offers a number of advantages regarding architecture exploration, simulation performance, and early software development. The tendency is therefore to use TLM-based descriptions of SoC platforms as golden models that, by essence, must be flawless.

In this paper, a SoC critical embedded platform under development by Astrium is used as proof-of-concept demonstrator, to assess the ISIS prototype tool which is devoted to the verification of SystemC TLM designs. Given temporal properties that capture the intended requirements, ISIS automatically instruments the design with ad hoc checkers that inform about the satisfaction of the properties during simulation.

After a description of the target platform design, we show that the PSL language enables the unambiguous expression of the required properties, and that the checkers produced by ISIS verify their satisfaction with a limited simulation time overhead.

1 Introduction

As the complexity of Systems on Chips (SoC's) drastically increases, the need for new design methodologies is compelling, and the interest in languages like SystemC [1] is growing (SystemC is in fact a library of C++ classes for modeling electronic circuits). SystemC TLM (*Transaction Level Modeling*) [10] favors design reuse, architecture exploration, and early software development [4]. Also, due to its high level of abstraction, in particular for specifying the communication of complex data types between the components of the SoC, the simulation of TLM models is several orders of magnitude faster than RTL (*Register Transfer Level*) simulation, thus considerably improving productivity in SoC design [15]. Hence it is widely being adopted, and TLM specifications tend to become golden reference models [11] that, by essence, must be completely flawless. To that goal, Assertion-Based Verification (ABV) brings an attractive solution.

ABV addresses the issue of verifying that the design obeys a given collection of temporal assertions (or properties). Those assertions, written in languages such

* This work is supported by the French project SoCKET (FUI).

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 21–36, 2011.

as the IEEE standards PSL [3] and SVA [2], are used to capture the desired characteristics. They provide a way to drive the formal analysis of the design model. This is now a well-established technology *at the RT level* [19]. The leading CAD companies have integrated property checking in their RTL simulators; e.g., *ModelSim* (Mentor Graphics), *VCS* (Synopsys) and the *Incisive* platform of Cadence allow to complement VHDL or Verilog descriptions with temporal assertions to be checked during simulation. *No equivalent solution* exists for SystemC TLM descriptions, which are algorithmic specifications that make use of elaborate communication models for complex data types ("transactions") and communication interfaces.

ISIS [23,9] is an academic tool that actually answers the need for ABV at the system level. Given PSL assertions that express the intended behaviour, it automatically instruments the SystemC TLM design with ad hoc checkers. During simulation, those checkers provide information about the satisfaction of the assertions. The original simulation testbenches can be used, there is no need of specific ones. In the context of TLM descriptions, assertions of interest mainly target the verification of the hardware/software interoperability in the interactions on the SoC. They express properties regarding communications i.e., properties associated with transactional events (for instance, data are transferred at the right place in memory, a transfer does not start before the completion of the previous one, etc.). ISIS enables the verification of those kinds of requirements, and is usable with timed or untimed models. This paper illustrates its applicability to a representative SoC platform developed by Astrium.

The SoCKET project[1] gathers industrial and academic partners to address the issue of design methodologies for *critical* embedded systems. At different phases of the SoCKET design flow (see Fig. 1) [18], the satisfaction of dedicated properties has to be guaranteed (namely at the System level, and after HW/SW partitioning). Some of these requirements, originally provided as textual descriptions, can be fully disambiguated when translated into PSL assertions.

The ultimate goal of Astrium is to verify these properties at the System Level using TLM-based Virtual Platforms, but also at the implementation level using RTL designs. The ISIS tool fits the Functional Validation and SW Performance Validation steps.

In that framework, Astrium is developing a toolbox in SystemC dedicated to architecture prototyping and to benchmarking architecture performance. The prototype platform is an assembly of software and hardware models. Following the standard test process, we usually run one or more test scenari and check the results. This procedure is useful, but its outcome is only a measure of the quality of the results. It does not ensure that there is no unexpected side effect. The ISIS approach gives the possibility to complement this process by automatically instrumenting the code with functional properties in order to check them while executing a test scenario (e.g., to verify that a client does not read data in a DMA destination area when a transfer is on-going).

[1] *SoC toolKit for critical Embedded sysTems*, see `http://socket.imag.fr/`

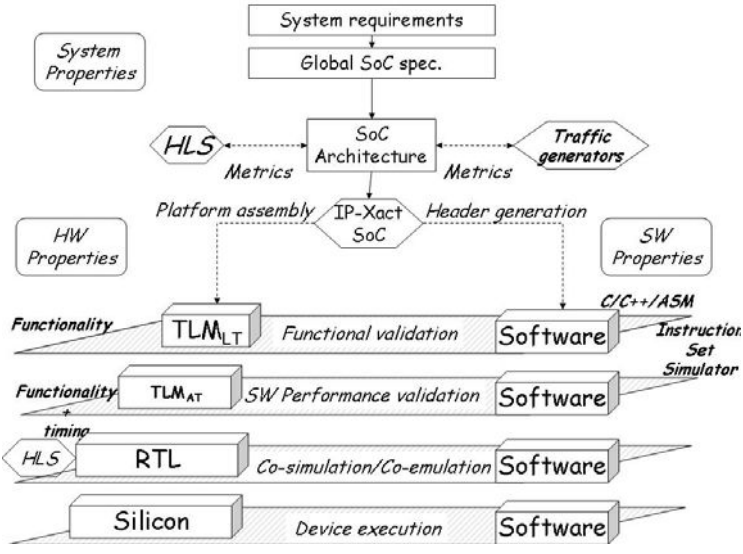


Fig. 1. SoCKET design flow [18]

2 ABV for SoC Platforms

2.1 Expression of Requirements for TLM Designs

Brief Overview of PSL. The core of the PSL (*Property Specification Language*) language is the *Temporal layer* that gives the possibility to describe complex temporal relations, evaluated over a set of evaluation cycles. Basic expressions of the *Boolean layer* are used by the other layers. The *Modeling layer* is used to augment what is possible using PSL alone, in particular it allows to manage auxiliary (global) variables. The *Boolean* and *Modeling* layers borrow the syntax of the hardware description language in which the PSL assertions are included (SystemC, or more generally C++, in our case).

Formulas of the FL (Foundation Language) class of the PSL Temporal layer essentially represent linear temporal logic. Their semantics is defined with respect to *execution traces* (we will see thereafter that these traces are obtained by different discretizations depending on the abstraction level).

One key basic operator of the FL class is *strong until*, denoted $\mathsf{until!}$ or U: roughly speaking, $\varphi \,\mathsf{until!}\, \psi$ holds iff there exists an evaluation point in the trace from which ψ holds, and φ holds until that point.

The PSL formula $\mathsf{always}\,\varphi$ means that φ must be verified on each evaluation point of the trace.

Using the strong $\mathsf{next\_event!}$ operator, formula $\mathsf{next\_event!}(b)(\varphi)$ requires the satisfaction of φ the next time the boolean expression b is verified.

Finally, formula $\varphi \,\mathsf{before!}\, \psi$ means that φ must be satisfied before the point from which ψ holds.

Classically in temporal logics, the weak until operator until does not impose the occurrence of ψ. Weak versions of other operators (such as weak next_event and before) are derived accordingly. See [3] for more details about PSL.

Specifying at the System Level. At the Register Transfer level, which is close to the circuit implementation, properties of interest target a high level of accuracy, with detailed precisions of the behaviour of hardware elements (signals) with respect to the synchronization clock. Execution traces are usually built by sampling the simulation traces at clock ticks.

At the transactional (TLM) level, there is a need for addressing more coarse-grained properties, typically related to abstract communication actions. Moreover, there is no synchronization clock, and there is even often no explicit notion of time. An illustrative example of a simple assertion that can be expressed at this level is: *the intended address is used when a memory transfer occurs.*

At this level, ISIS[2] keeps the PSL semantics unchanged but observation points are when the assertion needs to be re-evaluated i.e., each time a variable of the assertion may be modified (an appropriate action occurs in the corresponding channel). Hence the execution traces are built by sampling at the communication actions that are related with the variables involved in the assertion.

Expressing Conditions on Communications. ISIS gives the user the possibility to express conditions on communication actions: not only the occurrence of a given communication, but also conditions on the values of the actual parameters of the corresponding function call (or on its return value).

With the various features offered by the TLM-1 and TLM-2 communication models, it may be important to differentiate the simple occurrence of a communication action, the start, or the end of a communication action. To that goal, ISIS automatically generates ad hoc predicates `fctname_CALL()`, `fctname_START()`, and `fctname_END()`, that can be used in Boolean expressions:

- `name.fctname_CALL()` and `name.fctname_START()` denote that the communication function `fctname` of the element `name` has just been called,
- and `name.fctname_END()` expresses that the communication function `fctname` of the element `name` just ended.

Complementarily, `name.fctname.p#` denotes the parameter in position `#` of function `fctname` (0 is used for the return value).

The Modeling layer of PSL allows declaring auxiliary variables (variables that are not part of the design) as well as giving behavior to them. It is of great interest in the TLM context, as will be demonstrated in section 4.3. Since the reference manual does not provide details about its semantics (it is simply commonly admitted that the statements of the Modeling layer should be evaluated at each step of the evaluation of the property), its implementation in ISIS has required the definition of an operational semantics [8].

[2] See `http://tima.imag.fr/vds/Isis/`

Example. Let us close this discussion with a simple example of platform, delivered with the first draft of the TLM 2.0 library: a master programs a DMA through a memory-mapped router to perform transfers between two memories. A property of interest is: *any time a source address of the first memory is transferred to the DMA, a read access in this memory eventually occurs and the right address is used* (a similar property can be used for the other memory).

The expression of this property requires the memorization of the source address at the moment it is transferred to the DMA (*write* operation on the initiator port of the master, to the destination of the DMA source register), to be able to check that this address is actually used when the memory is read (*read* operation on the first memory). The PSL Modeling layer is necessary: an auxiliary variable `req_src_addr` memorizes the source address (the second parameter of the *write* operation, expressed as `initiator_port.write.p2` in the ISIS formalism) each time the source register of the DMA is overwritten. Another extra variable `dma_src_reg` is used, but is just a constant that stores the address of the DMA source register, for the sake of assertion readability:

```
// ---- Modeling layer ----
// HDL_DECLs :
int req_src_addr;
int dma_src_reg = 0x4000+pv_dma::SRC_ADDR;
// HDL_STMTs :
// Memorization of the source address when transferred to the DMA:
if (initiator_port.write_CALL() && initiator_port.write.p1 == dma_src_reg)
    req_src_addr = initiator_port.write.p2;
// ---- Assertion ----
// PROPERTY :
assert always ((initiator_port.write_CALL() &&
                initiator_port.write.p1 == dma_src_reg &&
                initiator_port.write.p2 < 0x100)
        => next_event!(mem1.read_CALL())(mem1.read.p1 == req_src_addr));
```

2.2 Platform Instrumentation

To build property checkers from PSL assertions, ISIS uses a variant of the original interconnection method developed as the HORUS technology for RTL designs [22]: checkers are built compositionally from a library of elementary components that correspond to the primitive PSL operators. In the ISIS context, those elementary components are SystemC modules.

Moreover, ISIS provides a framework for the observation of the communication actions used to sample the simulation traces (as explained in section 2.1) [23]. Thus, starting from PSL assertions:

- the tool performs the automatic construction of the corresponding checkers,
- then the SystemC code of the design under verification is automatically instrumented to relate the variables involved in the properties (formal variables) to actual components of the design. For instance, in the example of section 2.1, the variables named `initiator_port` and `mem1` in the assertion

statement are actually the initiator port of the master and the first memory; this correspondance is provided by the user[3]. This allows for flexibility and, if required, reusability of the same assertion in different contexts.

The monitors are thus linked to the design under test through the observation mechanism, and it remains to run the SystemC simulator on the system made of this combination of modules, using the original testbenches. Any property violation during simulation is reported by the monitors. To the best of our knowledge, no other tool offers a comparable solution.

2.3 Related Works

Few results have been proposed regarding static or dynamic verification of SystemC TLM specifications. The static approaches proposed in [13], [20] and [14] concentrate on *model checking* SystemC designs. They consider either clock-synchronized descriptions ([13]) or actual TLM specifications ([20], [14]). Abstraction techniques are required to get tractable models, or only limited (pieces of) designs can be processed.

A first proposal for constructing checkers for SystemC designs was described in [12], but was restricted to clocked designs and to assertions that are roughly of the form *a condition holds at a given time point* or *during a given time interval*.

A simulation-based methodology to apply ABV to TL models is proposed in [21]. During a first simulation run, transaction traces are recorded in a VCD file. This file is then translated into a Verilog description that is used for a second simulation inside an RTL simulator, and SVA assertions can be checked at this level. This solution is necessarily time-consuming since two simulation runs are required. Moreover, transactions are simply mapped to Boolean signals (*true* during the transaction, and *false* otherwise), which prevents from considering the parameters of the communications.

In [6] the authors explicitly express events through a specialized extension of SVA. A framework to perform runtime verification of transactional SystemC designs is proposed, and a possible implementation using proxies is also mentioned in [7]. This approach is appealing, but both the additional semantics constructs and the framework appear unnecessarily complicated. A few experimental results are reported in [7]; even if they cannot be directly compared to those of ISIS since case studies are different, the simulation time overhead appears to be significant.

Lahbib uses the PSL property checkers generated by the FoCs tool to monitor PSL assertions in transactional designs [17,16]. The key feature of this work is the use of specific channel classes, containing instances of the property checkers. During simulation, the TLM method *transport* calls on the checker's transition function when a transaction is initiated. With this approach, the checkers are

[3] ISIS can be used with a graphical user interface; in that case the information is provided through a selection list. There also exists a scripting-oriented version, in which the user provides the information by means of attributes in an XML file.

enclosed inside the channels and this induces considerable limitations: the assertions cannot involve several channels and hybrid properties (i.e., including both signals and TLM channels) are not supported.

Some commercial tools also provide for introducing PSL assertions in SystemC designs. The tool *Cadence Incisive Unified Simulator* (v 6.11) supports TLM, but only signals can be involved in the assertions, and neither the next_event operator nor the strong temporal operators can be used. Other tools like *Synopsys VCS* only accept RTL descriptions.

To our knowledge, ISIS is the only existing tool with the following features:

- actual complex TLM descriptions are supported,
- the statements of the assertions can involve several channels, and this group of channels can be heterogeneous (signals and TLM channels),
- communication parameters can be taken into account,
- using the PSL Modeling layer, auxiliary variables can be used for the expression of the specifications.

Moreover it provides a high level of automation and it is quite efficient, both for the construction of the checkers and during instrumented simulation.

3 The Case Study

3.1 Platform Description

In the SoCKET project, Astrium proposes a use case in the Guidance/Navigation/Control domain, specified as an image processing algorithm supporting mobile object extraction/tracking for overall telemetry compression. The main goal is to define the optimal data processing architectures for different sets of parameters of this algorithm.

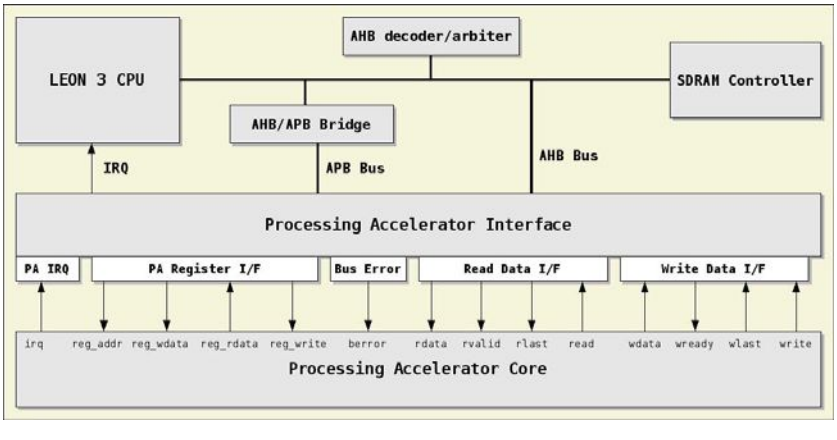


Fig. 2. Architecture block diagram: processor + dedicated HW accelerator

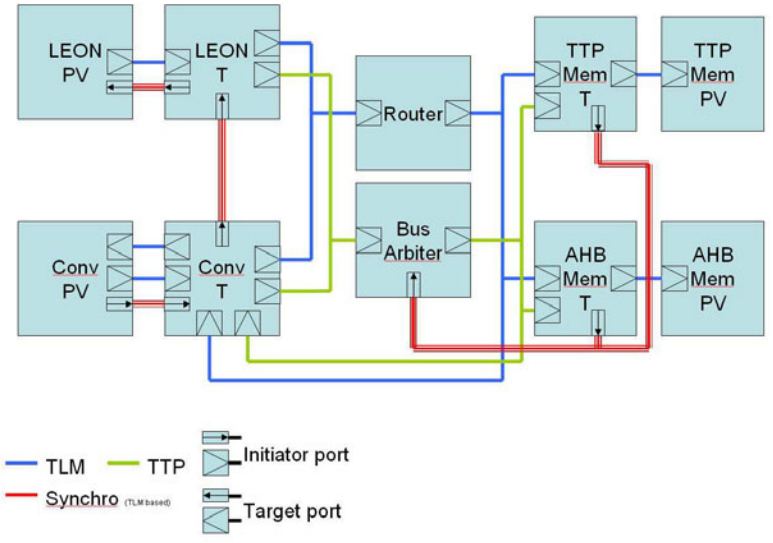


Fig. 3. Astrium demonstration platform

Various SoC architectures are targeted. In the first one, described in Fig. 2, the full algorithm is partioned into SW and HW pipelined stages: each HW accelerator is in charge of a complete algorithm step. The currently available SystemC platform (Fig. 3) gives a simplified but fully representative view of this architecture, where the accelerator function in a convolution operator. This platform models the functional behaviour, but also the timing constraints by means of an Astrium implementation of a technology for the separation of time and functionality in timed TLM modelling [5][4]. The ISIS experimentations have been performed using that platform.

The LeonPV is a pseudo-processor that emulates the behaviour of the target software ("PV" is the functional behaviour, "T" is the time layer that adds time annotation to the exchange). The Router routes each TLM transaction to one of the slave components, among them the TTP Mem (a memory) with its layer of time annotation (refresh, latency, ...). In parallel, a transaction is sent to the Bus which portrays the bus fabric characteristics (arbitration, latency, ...). Finally, the Conv component is also equipped with a time layer. This Conv IP is a convolution block able to fetch and store data in the main memory without any processor intervention. It includes a register file with Command Register and DMA configuration register. In the ultimate platform, the pseudo-processor will be replaced by an Instruction Set Simulator (ISS) running the definitive embedded code.

[4] TTP = Timed TLM Protocol.

3.2 Some Typical Requirements

Various types of properties can be considered for this kind of platform. They can be classified into specific categories, described below. A representative property of each category has been selected for verification with the ISIS technology.

Software Synchronisation Rules. These rules are related to the correct operation of the software with respect to the hardware e.g., answer to an event response time, concurrent access, buffer under/overflow.
Platform rule 1: The processor does not start a new convolution processing before the completion of the previous one.

Coding Constraint Rules. These rules check that the coding constraints imposed by the quality rules or the hardware requirements are respected e.g., registers are programmed before starting an operation, a given sequence is obeyed when using a block.
Platform rule 2: No convolution processing must be started before both destination and source addresses have been programmed.

Architectural Constraint Rules. These rules check that architectural properties are respected by the software e.g., an AMBA target retry-enable is only accessed by one master, flash memory write state machine is well managed. They can also check some temporal properties related to the overall system behaviour like arbitration policy or maximum latency constraints.
Platform rule 3: The memory does not respond with two "splits" consecutively to the same master (the bus does not support multiple split transactions; the system architecture must prevent this behaviour).

4 Application of ABV to the Case Study

Nowadays, the Astrium's validation process is mainly based upon the development and execution of dedicated tests: specific tests are developed for each requirement, and each test addresses a given requirement. Well-suited sets of tests have to be tailored for each validation phase in the design flow (at IP level, sub-system level and top level). The approach of assertion-based verification should instead enable the formal description of properties at system level, to be refined and reused later on. This should reduce the specification effort and lead to safer verifications. The solution described in section 2 provides the starting point of such an ABV flow.

We illustrate here the applicability of this solution to the platform and requirements of section 3. That also gives us the occasion to express remarks about the facets of ABV at the system level. In addition to the DMA example of section 2.1, the capabilities of ISIS have been demonstrated on a variety of case studies, ranging from small systems with FIFO, faulty channel, packet switch, to a Motion-JPEG decoding platform [8,9,24]. The experiments reported here regarding the Astrium's platform show that the approach is workable in an industrial context.

4.1 Convolution Processings in Sequence

The first property states that *the Leon processor does not start a new convolution processing before the end of the previous one*. To formalize this property, it is necessary to identify the designer's view of starting and ending a convolution processing. From a hardware-oriented point of view, starting a processing is identified by writing in the read address register of the convolution unit (denoted `a_read_addr`), and the end of a processing corresponds to reading in the length register of the convolution unit (denoted `a_write_length`) a value that equals image_size. The PSL formalization of this property is given below (it also takes into account the fact that this assertion should hold only when `PVT_NO_CHECK`, which is a particular debugging flag, is deactivated). The declarative part of the Modeling layer is simply used to declare constants for the addresses of the registers and for the image size. The assertion part states that, every time the processor starts a new convolution processing (writes into `a_read_addr`), the end of this processing (read image_size in `a_write_length`) will occur before the next start.

```
vunit prop1 {
   // HDL_DECLs :
   unsigned int l=NBL, c=NBC;
   unsigned int image_size = l * c;
   unsigned int hw_conv_address = ttp_validation::get_address_map(HW_CONV);
   unsigned int a_read_addr = hw_conv_address +
                              ttp_validation::conv_reg::a_read_addr;
   unsigned int a_write_length = hw_conv_address +
                              ttp_validation::conv_reg::a_write_length;
   unsigned int NO_CHECK = prt_tlm_ttp::PVT_NO_CHECK;
   // HDL_STMTs :
   // no statement here.
   // PROPERTY :
   assert
      always(leonPVinitiator_port.write_CALL()
               && leonPVinitiator_port.write.p1 == a_read_addr
               && leonPVinitiator_port.write.p5 != NO_CHECK
        => next ( (leonPVinitiator_port.read_END()
                    && leonPVinitiator_port.read.p1 == a_write_length
                    && leonPVinitiator_port.read.p2 == image_size
                    && leonPVinitiator_port.read.p5 != NO_CHECK)
                  before (leonPVinitiator_port.write_CALL()
                          && leonPVinitiator_port.write.p1 == a_read_addr
                          && leonPVinitiator_port.write.p5 != NO_CHECK)));
}
```

We can remark that this assertion refers to a variable *leonPVinitiator_port*. As explained in section 2.2, the user indicates that this variable is actually the initiator port of the Leon processor in the platorm, which corresponds to the element called *Top.m_leon.m_leon_PV.initiator_port* in the SystemC code.

4.2 Convolution Processing and Transfer of Addresses

The second property states that *both the destination and source addresses must be sent to the convolution unit before it starts a convolution processing.* Here too, this assertion should hold only when `PVT_NO_CHECK` is deactivated. As already mentioned, starting a processing is identified by writing in the read address register of the convolution unit (denoted `a_read_addr`); its destination address register is denoted `a_write_addr`. The assertion expresses that, for each processing, the destination address must be transmitted before the source address (sending the source address is the last action, that triggers the convolution processing). The assertion is made of the conjunction of two sub-assertions: the first one is related to the very first processing, and the second one checks the expected behaviour for the other processings (a new processing is recognized by the end of the previous one). Here too, *leonPVinitiator_port* is *Top.m_leon.m_leon_PV.initiator_port.*

```
vunit prop2 {
   // HDL_DECLs :
   unsigned int l=NBL, c=NBC;
   unsigned int image_size = l * c;
   unsigned int hw_conv_address = ttp_validation::get_address_map(HW_CONV);
   unsigned int a_read_addr = hw_conv_address +
                              ttp_validation::conv_reg::a_read_addr;
   unsigned int a_write_addr = hw_conv_address +
                              ttp_validation::conv_reg::a_write_addr;
   unsigned int a_write_length = hw_conv_address +
                              ttp_validation::conv_reg::a_write_length;
   unsigned int NO_CHECK = prt_tlm_ttp::PVT_NO_CHECK;
   // HDL_STMTs :
   // no statement here.
   // PROPERTY :
   assert
        ((leonPVinitiator_port.write_CALL()
           && leonPVinitiator_port.write.p1 == a_write_addr
           && leonPVinitiator_port.write.p5 != NO_CHECK)
         before (leonPVinitiator_port.write_CALL()
                && leonPVinitiator_port.write.p1 == a_read_addr
                && leonPVinitiator_port.write.p5 != NO_CHECK))
        &&
        always((leonPVinitiator_port.read_END()
              && leonPVinitiator_port.read.p1 == a_write_length
              && leonPVinitiator_port.read.p2 == image_size
              && leonPVinitiator_port.read.p5 != NO_CHECK)
          => next((leonPVinitiator_port.write_CALL()
                    && leonPVinitiator_port.write.p1 == a_write_addr
                    && leonPVinitiator_port.write.p5 != NO_CHECK)
                  before (leonPVinitiator_port.write_CALL()
                           && leonPVinitiator_port.write.p1 == a_read_addr
                           && leonPVinitiator_port.write.p5 != NO_CHECK)))
}
```

4.3 Successive Splits Forbidden

The third property aims at verifying that *the AHB memory cannot emit two successive splits for the same master*. This assertion is simple and intuitive, but is challenging regarding its formalization. Indeed, a more precise statement of the property is: if a master accesses the AHB memory (read or write operation) and the status of this communication indicates that a split has been issued, then there must be no split when the master retries its access. Here, the Modeling layer is mandatory to memorize the id of the last master that accessed the AHB memory. We also use it to manage a Boolean variable `status_split`.

Depending on the objectives of the designer, it may be decided to which extent the AHB bus is also concerned with this assertion: the communication can be observed on the initiator port of the AHB bus (version 1 below), or on the slave port of the AHB memory (version 2 below).

```
vunit prop3_v1 {
    // HDL_DECLs :
    unsigned int prev_master, master = 999;
    bool to_ahb;   // true if the target is the AHB memory (target 0)
    bool status_split;  // true if a split has been issued
    prt_tlm_ttp::ttp_response<ttp_ahb::ahb_status> resp;
    prt_tlm_ttp::ttp_status<ttp_ahb::ahb_status> s;
    // HDL_STMTs :
    if (bus_initiator_port.do_transport_END()) {
        to_ahb = (bus_initiator_port.do_transport.p3 == 0);
        // if the target of the communication is the AHB memory:
        if (to_ahb) {    // set ``master'' and ``status_split''
            prev_master = master;
            master = bus_initiator_port.do_transport.p1.get_master_id();
            resp = bus_initiator_port.do_transport.p2;
            s = resp.get_ttp_status();
            status_split = (s.access_extension())->is_split();
        }
    }
    else {  to_ahb = false;
            status_split = false;
    }
    // PROPERTY :
    assert always((bus_initiator_port.do_transport_END()
                   && to_ahb && status_split)
              => next (next_event(bus_initiator_port.do_transport_END()
                                  && to_ahb && (master == prev_master))
                                 (!status_split)));
}
```

where *bus_initiator_port* is in fact *Top.m_bus.initiator_port*. The second version is similar, but directly observes the `transport` calls for the memory (the Boolean variable `to_ahb` is no more required):

```
vunit prop3_v2 {
    // HDL_DECLs :
    unsigned int prev_master, master = 999;
    bool status_split;  // true if a split has been issued
    prt_tlm_ttp::ttp_response<ttp_ahb::ahb_status> resp;
    prt_tlm_ttp::ttp_status<ttp_ahb::ahb_status> s;
    // HDL_STMTs :
    if (ahb_mem.transport_END()) { // set ‘‘master’’ and ‘‘status_split’’
         prev_master = master;
         master = ahb_mem.transport.p1.get_master_id();
         resp = ahb_mem.transport.p0;
         s = resp.get_ttp_status();
         status_split = (s.access_extension())->is_split();
    }
    else status_split = false;
    // PROPERTY :
    assert always((ahb_mem.transport_END() && status_split)
                 => next (next_event(ahb_mem.transport_END()
                                     && (master == prev_master))
                                    (!status_split)));
}
```

and *ahb_mem* here is the slave *Top.m_memory_with_split.m_ahb_memory_T*.

4.4 Performances

CPU times for these experiments demonstrate that the ISIS monitoring technology incurs moderate overhead in simulation time. In Table 1, CPU times are taken on an Intel Core2 Duo (3 GHz) under Debian Linux. The first column recalls the simulation context:

– for properties 1 and 2, simulations check image processing by means of the convolution block. The testbench is configured to process 100000 images. Here we simply performed untimed simulations, but comparable results are obtained in the case of timed simulations.

Table 1. Experimental results

	Simulation	Monitoring		CPU time overhead	Number of property evaluations
		No	Yes		
P1	Processing of 100000 images with the convolution block (untimed)	9.38 s	9.81 s	+4.6%	500000
P2	Processing of 100000 images with the convolution block	9.38 s	10.02 s	+6.8%	500000
P3 v1	Processing of 30000 images by the Leon itself (timed)	7.68 s	8.34 s	+8.6%	600000
P3 v2	Processing of 30000 images by the Leon itself (timed)	7.68 s	7.83 s	+2%	360000

– for property 3, timed simulations check image processing by the Leon, with storage of the resulting images in the AHB memory. The testbench is configured to process 30000 images.

The second column gives SystemC simulation times without any monitoring. The third column gives CPU times for simulations while monitoring with the checkers constructed by ISIS. Checkers construction and code instrumentation times are negligible. The number of times the checker functions have been evaluated during those simulations is reported in the fifth column.

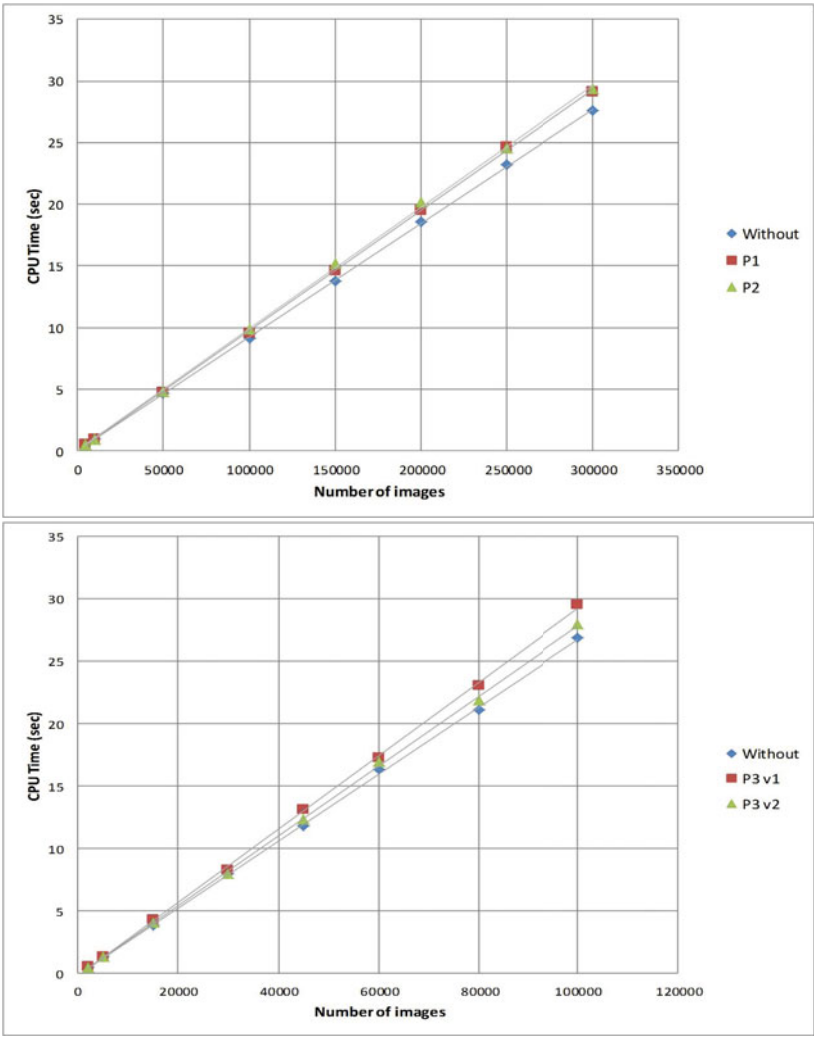


Fig. 4. Simulation times for various amounts of processed images

Figure 4 shows that the percentage of CPU time overhead remains the same whatever the amount of images processed by the platform. The first chart compares CPU times for simulations without any monitoring and with the monitoring of properties 1 and 2, for 50000 to 300000 images. The second chart compares CPU times for simulations without any monitoring and with the monitoring of the two versions of property 3, for 15000 to 100000 images.

5 Conclusion

After these experiments, Astrium issued a first noticeable return of experience, summarized as follows:

- the expressivity of PSL is fine, and the benefits provided by the Modeling layer are undoubtable,
- the moderate overhead induced by the ISIS monitors is attractive,
- these monitors will enable non-regression testing when introducing new versions of the platform components, such as the Leon instruction set simulator,
- there is a need to define strict rules for the textual (natural language) description of the properties. As a preliminary proposal:
  - to respect the TLM "philosophy", specify at TLM interfaces only (module boundaries only),
  - to enable the formalization of the properties, the model developer must explictly specify the observation/connection points and disambiguate the meaning of transactions in terms of function calls and parameters involved (e.g., in property 1, *starting a processing* is identified by writing in the read address register of the convolution IP).

Our future works include the refinement of these property definition rules. We will also focus on the relation between such PSL properties at the TLM level and their counterparts at the RT level, as well as on prototyping the embedding of the automatically generated monitors into the final design in order to mitigate the space radiation environment effects at the architecture level (today, most of the mitigation is performed at the silicon technology level).

Acknowledgments. The authors are grateful to A.Berjaoui for his help with the SystemC encoding.

References

1. IEEE Std 1666-2005, IEEE Standard System C Language Reference Manual. IEEE (2005)
2. IEEE Std 1800-2005, IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language. IEEE (2005)
3. IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL). IEEE (2005)
4. Avinun, R.: Validate hardware/software for nextgen mobile/consumer apps using software-on-chip system development tools. EETimes (December 2010), `http://www.eetimes.com/design/embedded/4211507/Validate-hardware-software-for-nextgen-mobile-consumer-apps-using-software-on-chip-system-development-tools-`

5. Cornet, J.: Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip. PhD thesis, INP Grenoble (2008)
6. Ecker, W., Esen, V., Hull, M.: Specification Language for Transaction Level Assertions. In: Proc. HLDVT 2006 (2006)
7. Ecker, W., Esen, V., Hull, M.: Implementation of a Transaction Level Assertion Framework in SystemC. In: Proc. DATE 2007 (2007)
8. Ferro, L., Pierre, L.: Formal Semantics for PSL Modeling Layer and Application to the Verification of Transactional Models. In: Proc. DATE 2010 (March 2010)
9. Ferro, L., Pierre, L.: ISIS: Runtime Verification of TLM Platforms. In: Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's. LNEE, vol. 63. Springer, Heidelberg (2010)
10. Ghenassia, F. (ed.): Transaction-Level Modeling with SystemC. Springer, Heidelberg (2005)
11. Goering, R.: Transaction models offer new deal for EDA. EETimes (March 2006), http://www.eetimes.com/showArticle.jhtml?articleID=181503693
12. Große, D., Drechsler, R.: Checkers for SystemC Designs. In: Proc. ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2004) (2004)
13. Habibi, A., Tahar, S.: Design and Verification of SystemC Transaction Level Models. IEEE Transactions on VLSI Systems 14(1) (January 2006)
14. Karlsson, D., Eles, P., Peng, Z.: Formal Verification of SystemC Designs Using a Petri-Net Based Representation. In: Proc. DATE 2006 (2006)
15. Klingauf, W., Burton, M., Günzel, R., Golze, U.: Why We Need Standards for Transaction-Level Modeling. SOC Central (April 2007)
16. Lahbib, Y., Perrin, A., Maillet-Contoz, L., Clouard, A., Ghenassia, F., Tourki, R.: Enriching the Boolean and the Modeling Layers of PSL with SystemC and TLM Flavors. In: Proc. FDL 2006 (2006)
17. Lahbib, Y.: Extension of Assertion-Based Verification Approaches for the Verification of SystemC SoC Models. PhD thesis, Univ. of Monastir, Tunisia (2006)
18. Lefftz, V., Bertrand, J., Cassé, H., Clienti, C., Coussy, P., Maillet-Contoz, L., Mercier, P., Moreau, P., Pierre, L., Vaumorin, E.: A Design Flow for Critical Embedded Systems. In: Proc. IEEE Symposium on Industrial Embedded Systems (SIES) (July 2010)
19. Jon Michelson and Faisal Haque. Assertions Improve Productivity for All Development Phases. EETimes (July 2007), http://www.eetimes.com/design/edadesign/4018491/Assertions-Improve-Productivity-for-All-Development-Phases
20. Moy, M., Maraninchi, F., Maillet-Contoz, L.: LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. Design Automation for Embedded Systems (2006)
21. Niemann, B., Haubelt, C.: Assertion-Based Verification of Transaction Level Models. In: Proc. ITG/GI/GMM Workshop (February 2006)
22. Oddos, Y., Morin-Allory, K., Borrione, D.: Assertion-Based Design with Horus. In: Proc. MEMOCODE 2008 (2008)
23. Pierre, L., Ferro, L.: A Tractable and Fast Method for Monitoring SystemC TLM Specifications. IEEE Transactions on Computers 57(10) (October 2008)
24. Pierre, L., Ferro, L.: Enhancing the Assertion-Based Verification of TLM Designs with Reentrancy. In: Proc. MEMOCODE 2010 (2010)

Past Time LTL Runtime Verification for Microcontroller Binary Code

Thomas Reinbacher[1], Jörg Brauer[2], Martin Horauer[3], Andreas Steininger[1], and Stefan Kowalewski[2]

[1] Embedded Computing Systems Group, Vienna University of Technology, Austria
[2] Embedded Software Laboratory, RWTH Aachen University, Germany
[3] Department of Embedded Systems, UAS Technikum Wien, Austria

Abstract. This paper presents a method for runtime verification of microcontroller binary code based on past time linear temporal logic (ptLTL). We show how to implement a framework that, owing to a dedicated hardware unit, does not require code instrumentation, thus, allowing the program under scrutiny to remain unchanged. Furthermore, we demonstrate techniques for synthesizing the hardware and software units required to monitor the validity of ptLTL specifications.

1 Introduction

Program verification deals with the problem of proving that all possible executions of a program adhere to its specification. Considering the complexity of contemporary embedded software, this is a particularly challenging task. Conventional ad-hoc testing is significantly less ambitious; it is thus the predominant method in the (embedded) software industry. Typically, a set of test-cases is derived manually or automatically in a best effort fashion. Then, the arduous task of judging the results of a test-case run often remains with the test engineer.

1.1 Runtime Verification by Code Instrumentation

The field of runtime verification [7] has gained momentum as it links traditional formal verification and monitoring the execution of test cases. The aim is to increase confidence in correctness of the system, without claiming freedom from defects. In runtime verification, test oracles, which reflect the specification, are either automatically derived (e.g., from a given temporal logic formula that specifies a requirement) or formulated manually in some form of executable code. Correctness of an execution is then judged by means of evaluating sequences of events observed in an instrumented version of the program under scrutiny. Instrumentation can either be done manually, or automatically by scanning available program nodes (e.g., assignments, function calls, ...) at the level of the implementation language. Function calls are then inserted to emit relevant events to an observer, i.e., the test oracle. The latter approach has proven feasible for high-level implementation languages such as C, C++, and Java, as well as for hardware description languages such as VHDL and Verilog. Various runtime verification frameworks have thus emerged [8, 6, 13, 20, 14].

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 37–51, 2011.
Springer-Verlag Berlin Heidelberg 2011

1.2 Pitfalls of Code Instrumentation

Despite considerable technical progress, existing approaches to runtime verification are not directly transferable to the domain of embedded systems software, mainly due to the following reasons:

1. Embedded code often adopts target-specific language extensions, direct hardware register and peripheral access, and embedded assembly code. When instrumenting such a code basis, one has to take all the particularities of the target system into account, depleting the prospect of a universal approach.
2. In its present shape runtime verification proves the correctness of high-level code. However, to show that a high-level specification is correctly reproduced by the executable program, it is necessary to verify the translation applied to the high-level code as it is not unknown for compilation to introduce errors [9, 2, 24]. One thus needs to prove that for a given source code P, if the compiler generates a binary code B without compilation errors, then B behaves like P [29]. Proving correctness of the compiler itself is typically not feasible due its complexity and its sheer size [21, 22]. Flaws introduced by the compiler may thus remain unrevealed by existing approaches.
3. Instrumentation at binary code level is never complete as long as the full control flow graph (CFG) is not reconstructed from the binary program. Although CFG reconstruction of machine code is an active research area [3, 12, 17], generating sound yet precise results remains a challenge.
4. Instrumentation increases memory consumption, which may be of economical relevance for small-sized embedded targets.

We conclude that a non-instrumenting approach for microcontroller binary code may be a notable contribution to further establish the use of lightweight formal techniques, such as runtime verification, in the embedded software industry.

1.3 Requirements to Runtime Verification of Microcontroller Code

To overcome the pitfalls discussed so far, which prohibit the application of existing frameworks to the embedded systems domain, it is necessary to provide a framework that works on the level of binary code and additionally satisfies the following requirements:

Req1: Generality. For a verification on the binary code level the target microcontroller must be fixed. However, the approach shall not be bound to a certain compiler (version) or high-level programming language.

Req2: No Code Instrumentation. Typically, software event triggers are instrumented to report execution traces as sequences of observations. For small-scale embedded platforms, it is necessary to extract event sequences without code instrumentation.

Req3: Provide Mechanics to Evaluate Atomic Propositions. The atomic propositions (AP) of the specification need to be evaluated on microcontroller states of the running system. We need to find a reasonable trade-off between expressiveness of the AP and the complexity of their evaluation.

Furthermore, to be useful in an industrial environment, some practical requirements need to be considered:

Req4: Automated Observer Synthesis. From a user point of view, it is desirable to input a specification in some (temporal) logic, which is automatically synthesized into an observer that represents the semantics of the temporal property.

Req5: Usability. We aim at a framework which is applicable in industrial software development processes; at best, this is a push-button solution. It shall be possible to include implementation-level variables in the specification, which are automatically mapped to the memory state on the target hardware.

1.4 Contributions to Runtime Verification

The contribution of this paper is a framework for supervising past time linear temporal logic (ptLTL) properties [15] in embedded binary code. ptLTL allows to specify typical requirements to embedded software in a straightforward fashion, which contrasts with our experiences using Computation Tree Logic (CTL) [31]. Further, we present a host application that interacts with a customized hardware monitoring unit and a microcontroller IP-core (executing the software under scrutiny), both of which are instantiated within an FPGA. In our approach, supervision of ptLTL specifications can take place either *offline* (using the host application) or *online* in parallel to program execution. Both options come along without any kind of code instrumentation or user-interaction. We implemented the presented approach into our testing framework called CevTes [30].

1.5 Structure of the Paper

The presentation of our contributions is structured as follows. In Sect. 2, we present preliminaries used throughout the paper. Sect. 3 introduces our framework for runtime verification of binary code. We apply our approach to a real-life example in Sect. 4. We put our work in context with related work in Sect. 5 and conclude with a discussion of achievements in Sect. 6.

2 Preliminaries

This section introduces notations used in the remainder of the paper, including a formal microcontroller model and the finite-trace temporal logic ptLTL.

2.1 Formal Microcontroller Model

Addressing Memory Locations. Let $\mathsf{Addr} = \{0 \leq x < |\mathsf{Mem}| \ : \ x \in \mathbb{N} \cup \{0\}\}$ denote the set of memory locations of the microcontroller, where Mem represents the (linear) address space of the microcontroller memory. We write r_x to address a specific memory location, e.g. , r_{20} denotes the memory location with address 20. We assume a memory mapped I/O architecture (e.g. Intel MCS-51), thus, I/O registers reside within Mem.

State of the Microcontroller Program. In the following, let $\mathbb{N}_k = \{0, \ldots, k-1\}$. A state S of the microcontroller is a tuple $\langle pc, m\rangle \in \mathsf{Locs} \times (\mathsf{Addr} \to \mathbb{N}_{2^w})$, where Locs is a finite set of program counter values, and $m : \mathsf{Addr} \to \mathbb{N}_{2^w}$ is a map from memory locations (with bit-width w) to memory configurations. The state space of the program is thus a subset of $\mathsf{Locs} \times (\mathsf{Addr} \to \mathbb{N}_{2^w})$. We denote the initial microcontroller state S_0 by $\langle \texttt{0x00}, m_0\rangle$ where m_0 represents the configuration of all memory locations after power-up and `0x00` is the assumed reset vector.

State Updates. State updates trigger a state transition, thereby, transforming a predecessor state S^{-1} into the current state S. A state update is a triple $\delta = \langle \zeta_\delta, @_\delta, pc_\delta\rangle$, where ζ_δ is the new configuration of the altered memory location, $@_\delta$ is its address, and pc_δ is the new program counter value. Given a strict sequential execution of the program, state updates are in temporal order. A state update $S^{-1} \xrightarrow{\delta} S$ transforms $S^{-1} = \langle pc^{-1}, m^{-1}\rangle$ into $S = \langle pc_\delta, m\rangle$ where:

$$m(i) = \begin{cases} \zeta_\delta & \text{if } i = @_\delta \\ m^{-1}(i) & \text{otherwise} \end{cases}$$

A sequence of events, denoted π, is a trace of state updates δ, e.g. , $\pi = \langle \delta_0 \ldots \delta_n\rangle$.

2.2 Past Time LTL

While past time operators do no yield extended expressive power of future time LTL [10, Sect. 2.6], a specification including past time operators may sometimes be more natural to a test engineer [23, 19]. A ptLTL formula ψ is defined as

$$\begin{aligned} \psi ::= & \mathit{true} \mid \mathit{false} \mid \mathit{AP} \mid \neg\psi \mid \psi \bullet \psi \\ & \odot\psi \mid \diamond\psi \mid \boxdot\psi \mid \psi\ S_s\ \psi \mid \psi\ S_w\ \psi \end{aligned}$$

where $\bullet \in \{\wedge, \vee, \rightarrow\}$. $\odot\psi$ means *previously* ψ, i.e., it is the past-time analogue of next. Likewise, the other temporal operators are defined as: $\diamond\psi$ expresses *eventually in the past* ψ and $\boxdot\psi$ is referred to as *always in the past*. The duals of the until operator are S_s and S_w, i.e. , *strong since* and *weak since*, respectively.

Monitoring Operators. These basic operators can be augmented by a set of monitoring operators [15, 20]. The semantics of the monitoring operators is derived from the set of basic operators in ptLTL, thus, do not add any expressive power. However, they provide the test engineer a succinct representation of the most common properties emerging in practical approaches:

$$\psi ::= \uparrow \psi \mid \downarrow \psi \mid [\psi, \psi)_s \mid [\psi, \psi)_w$$

$\uparrow \psi$ stands for *start* ψ (i.e., ψ was false in the previous state and is true in the current state, equivalent to $\psi \wedge \neg \odot \psi$), $\downarrow \psi$ for *end* ψ (ψ was true in the previous state and is false in the current state, equivalent to $\neg\psi \wedge \odot\psi$), and

$[\psi_1, \psi_2)$ for *interval* ψ_1 ψ_2 (ψ_2 was never true since the last time ψ_1 was true, including the state when ψ_1 was true, equivalent to $\neg\psi_2 \wedge ((\odot\neg\psi_2)\ S\ \psi_1)$). The set of atomic propositions AP contains statements over memory locations in Locs. Space constraints force us to refer the reader to [15, 20, 10] for a formal semantics.

Determining Satisfaction. It is important to appreciate that satisfaction of a ptLTL formula can be determined along the execution trace by evaluating only the current state S and the results from the predecessor state S^{-1} [15].

3 System Overview

The following section details our runtime verification framework, as depicted in Fig. 1, which works on microcontroller binary code rather than a high-level representation of the program, thus meeting Req1.

We address Req2 by a hardware monitor unit, which is transparently attached to an industrial microcontroller IP-core running on an FPGA. The monitor allows to extract execution traces without code instrumentation. We tackle Req3 by a twofold approach: (i) Offline mode: We permanently send state updates

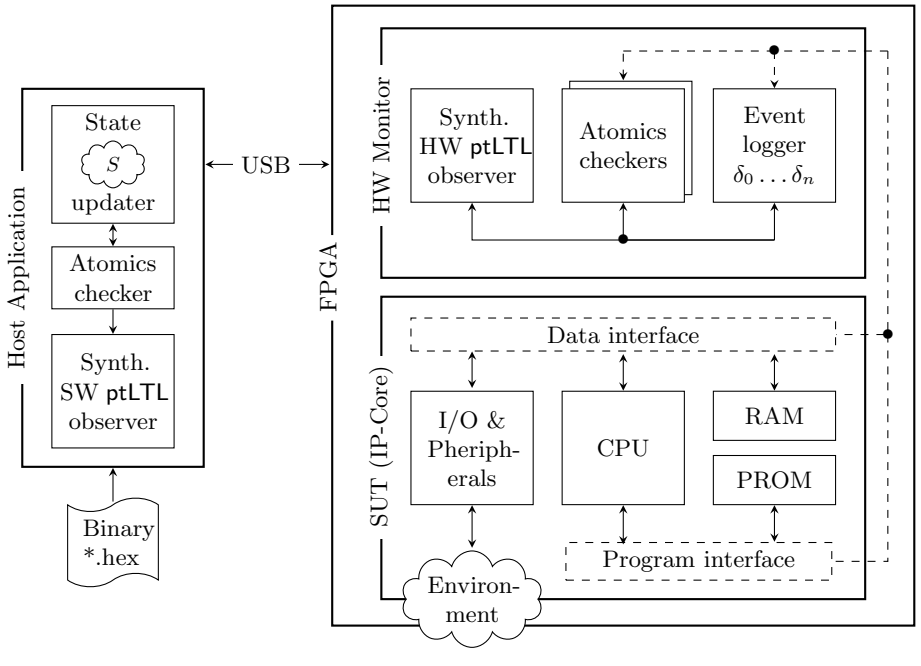


Fig. 1. System overview

δ from an hardware implemented event logger to a host application which applies δ to the current state S^{-1} to obtain the successor state S. The AP of the ptLTL formula ψ are evaluated on S and the validity of ψ is decided by a synthesized SW ptLTL observer. (ii) Online mode: As a self-contained alternative, we check the AP of the ptLTL formula on-the-fly and decide the validity of ψ by a synthesized HW ptLTL observer directly on the FPGA.

We meet Req4 by instantiating an algorithm described by Havelund and Roşu [15], i.e., we generate observer for ptLTL as executable Java or VHDL code. Finally, we comply with Req5 by providing a graphical interface to the system and an optional debug file parser allowing to state formulas over high level symbols.

3.1 ptLTL Observer Synthesis

Runtime verification requires an observer to be attached to the system under test. Our approach supports a full FPGA based solution as well as a combined one where a hardware event logger stimulates a Java class on the host computer. Technically speaking, we derive (a) Java classes and (b) VHDL entities, both representing an observer for the specification ψ. In a subsequent step, (a) is compiled into executable Java code and (b) is synthesized into a netlist. Both observers rely on the hardware monitor unit to evaluate the AP of ψ. Whereas (a) utilizes event updates about the state of the microcontroller, (b) makes use of a dedicated atomics checker hardware unit.

Observer synthesis thus consists of the following stages: (i) We use the ANTLR parser generator [26] to parse a ptLTL formula ψ, which yields an abstract syntax tree (AST) representing the specification. (ii) After some preprocessing of the AST, we determine the n subformulas $\psi_0 \dots \psi_n$ of ψ using a post-order traversal of the AST. (iii) We generate observers as executable Java or synthesizable VHDL code [15].

3.2 Hardware Monitor Unit

The hardware monitor unit (cf. Fig. 1) is attached to the system under test, an (unmodified) off-the-shelf microcontroller IP-core[1], which is embedded into its application environment. The observer consists of three main components, namely an event logger, an atomics checker unit, and a synthesized ptLTL observer. The remainder of this section discusses the details of these components.

Event Logger. The event logger wiretaps the data and the program interface of the microcontroller and collects memory updates δ non-intrusively. For example, if the currently fetched instruction is `MOV [*20, 0x44]`, which moves the constant value `0x44` into r_{20}, and the current program counter pc equals `0xC1C1`, then the event logger assembles a new state update $\delta = \langle \texttt{0x44}, 20, \texttt{0xC1C1} \rangle$.

[1] For our actual implementation we employ an Intel MCS-51 IP-core from Oregano Systems (`http://www.oregano.at`).

Atomics Checkers. The purpose of these units is to check the atomic propositions of ψ, one per unit. Ideally, we would favor a full-fledged hardware-only solution allowing for arbitrary atomic propositions to be checked on-the-fly. However, as we aim at a lightweight monitor with small area overhead, we opted for offering two implementation variants: a software-implemented offline checker supports arbitrary expressions for atomics, and we use constraints similar to Logahedra [16] for the hardware-based online approach, thus allowing to establish a balance between hardware complexity and expressiveness. More specifically, the hardware-based atomics checker supports conjunction of restricted two-variable-per-inequality constraints of the form

$$(\pm 2^n \cdot r_i \pm 2^m \cdot r_j) \bowtie C$$

where $r_i, r_j \in \mathsf{Mem}$, $C \in \mathbb{Z}$, $n, m \in \mathbb{Z}$, and $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. The second operand is optional, thus allowing range constraints of the form $\pm(2^n) \cdot r_i \bowtie C$.

Fig. 2 shows the generic hardware design to evaluate a single atomic proposition. The unit is connected to the data interface. We instantiate one such unit for each $ap \in AP$; the derived verdicts $atomic(0 \ldots |AP|)$ serve as input for the ptLTL observer. The *constant* C is loaded into the compare unit; *mode* constitutes control signals to determine the operation to be performed on the operands. The write-enable signal issued by the CPU triggers the atomics checker unit which stores the value on the data bus in a register iff the destination address equals i or j, respectively. The shifter unit supports multiplication and division by 2^n. The arithmetic unit is a full-adder, serving both as adder and subtracter. Observe that, when $\mathsf{Add}(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle, \mathsf{c})$ is a ripple carry adder for arbitrary length unsigned vectors $\langle \mathsf{a} \rangle$ and $\langle \mathsf{b} \rangle$ and c the carry in, then a subtraction of $\langle \mathsf{a} \rangle - \langle \mathsf{b} \rangle$ is equivalent to $\mathsf{Add}(\langle \mathsf{a} \rangle, \langle \overline{\mathsf{b}} \rangle, 1)$. Relational operators can be built around adders in a similar way [18, Chap. 6].

Synthesized HW ptLTL Observer. The synthesized ptLTL observer unit subsumes the verdicts of the diverse atomic checker units over the respective AP of ψ into a final decision $\pi \models \psi$. While in the offline mode this function is performed in software, a dedicated hardware block is needed for the online mode.

Housekeeping. The hardware monitor unit supports writing the `*.hex` file under scrutiny into the target system's PROM and handles communication tasks between FPGA and host application using a high-speed USB 2.0 controller.

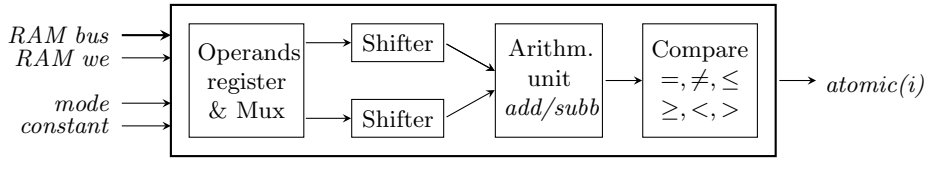


Fig. 2. The atomics checker unit

3.3 Host Application

The host application is responsible for offline runtime verification. It reads a `*.hex` binary file and a ptLTL formula ψ. Optionally, compiler-generated debug information is parsed and symbols in the high-level implementation language are related to memory addresses in microcontroller memory. Rather than expressing properties over memory locations within the RAM of the microcontroller, this approach allows high-level implementation symbols to be included in the formula. For example, the formula $\uparrow$ `foo` $= 20$ is satisfied iff the memory location that corresponds to the variable `foo`, say, r_{42}, does not equal 20 in the predecessor state S^{-1} and equals 20 in the current state S. Therefore, even though the analysis is based on binary code, it is possible to state propositions over high-level symbols, which eases the process of specifying desired properties.

State Updates. State transitions $S^{-1} \xrightarrow{\delta} S$ are performed on each state update δ, received from the event logger. Incoming events are categorized as follows: (i) events that perform plain state updates and (ii) events that alter memory locations used in atomic propositions of the formula. Events in (i) are used to keep a consistent representation of the current microcontroller state, whereas events in (ii) additionally trigger the SW ptLTL observer to derive a new verdict.

Event Evaluation. Atomic propositions are directly evaluated on the current state S, and the resulting verdicts are then forwarded to the observer that decides the validity of formula ψ.

Synthesized SW ptLTL Observer. Whenever the destination address @ of a state update δ matches any memory location in the atomic propositions AP of ψ, the generated software ptLTL observer code is executed and a new verdict is derived. If the property is violated, the unit reports "$\times$" to the user. State updates are in temporal order, thus, it would be possible to store a sequence $\langle \delta_1, \ldots, \delta_n \rangle$ of state updates and apply the observer afterwards, decoupled from program execution. However, in our experiments, it was always possible to evaluate the events without time-penalty, i.e., as they occur while the program is running. The stored state space consists only of the current state S.

4 Worked Example

In the remainder of this section, we report on applying our toolset to embedded C code. As an example, we consider a function block specified by the PLCopen consortium, which has defined safety-related aspects within the IEC 61131-3 development environment to support developers and suppliers of Programmable Logic Controllers (PLC) to integrate safety-related functionality into their systems. In the technical specification TC5 [28], safety-related function blocks are specified at a high level while the actual implementation is left to the application developer. The *emergency stop* function block [28, pp. 40 – 45], which we consider in the following, is a function block intended for monitoring an emergency stop button.

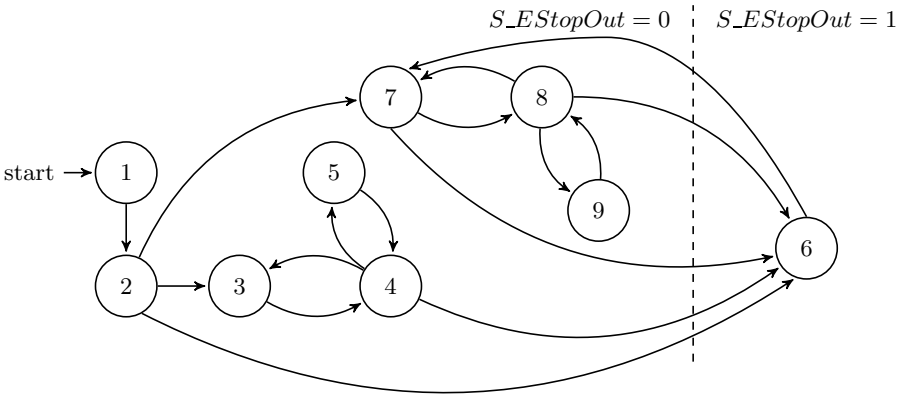


Fig. 3. The *emergency stop* function block as nondeterministic finite state machine

Interfaces. The function block senses five Boolean inputs, namely *Activate*, *S_EStopIn*, *S_StartReset*, *S_AutoReset*, *Reset* and drives three boolean outputs *Ready*, *S_EStopOut*, *Error* and one 16-bit wide diagnosis output *DiagCode*. *S_EStopOut* is the output for the safety-related response.

Requirements. The functional description of the block is given by PLCopen as a state diagram [28, p. 42]; Figure 3 shows a simplified version of the state machine (transition conditions and transitions from any state to the idle state (S_1) have been omitted to make the presentation accessible). Overall, the block comprises nine states, i.e., *Idle* (S_1), *Init* (S_2), *Wait for S_EStopIn1* (S_3), *Wait for Reset 1* (S_4), *Reset Error 1* (S_5), *Safety Output Enabled* (S_6), *Wait for S_EStopIn2* (S_7), *Wait for Reset 2* (S_8), and *Reset Error 2* (S_9).

Implementation. The implementation consists of approximately 150 lines of low level C code targeting the Intel MCS-51 microcontroller. For our experiments, we used the Keil μVision3 compiler. The compiled and linked `*.hex` file is written into the Intel MCS-51's PROM, serving as the system under test.

4.1 ptLTL Specification

In the implementation, the 8-bit unsigned variable *currState* represents the current state, of the function block. An enumeration maps the state numbers $\{S_1, \ldots, S_9\}$ to identifiers. To simplify presentation, we write Θ_{Sx} as abbreviation for the event $currState = S_x$. We proceed by describing two desired properties of the system.

Property 1. Predecessors of state *Safety Output Enabled* (S_6) are $\{S_2, S_4, S_7, S_8\}$, thus, S_6 shall not be reached from any other state, which is formalized as:

$$\psi^1 := \uparrow(\Theta_{S6}) \rightarrow \big[\uparrow(\Theta_{S2} \vee \Theta_{S4} \vee \Theta_{S7} \vee \Theta_{S8}),\ \uparrow(\Theta_{S1} \vee \Theta_{S3} \vee \Theta_{S5} \vee \Theta_{S9})\big)_S$$

The start of event Θ_{S6} implies that the start of $\{\Theta_{S2}, \Theta_{S4}, \Theta_{S7}, \Theta_{S8}\}$ was observed in the past; since then, the start of $\{\Theta_{S1}, \Theta_{S3}, \Theta_{S5}, \Theta_{S9}\}$ was never observed.

Property 2. Transitions to the reset states *Reset Error 1* (S_5) and *Reset Error 2* (S_9) shall only originate from *Wait for Reset 1* (S_4) and *Wait for Reset 2* (S_8), thus, have only a single predecessor state.

$$\psi^2 := \uparrow (\Theta_{S5}) \rightarrow \downarrow (\Theta_{S4})$$
$$\psi^3 := \uparrow (\Theta_{S9}) \rightarrow \downarrow (\Theta_{S8})$$

The start of event Θ_{S5} causes the end of event Θ_{S4}; the start of Θ_{S9} causes the end of Θ_{S8}.

4.2 Online Runtime Verification

We synthesized observers for properties ψ^1, ψ^2, and ψ^3 (cf. Fig. 5), both as VHDL hardware description and Java code. We sampled the emergency stop module with different, randomly-generated input patterns and could not find a property violation. To prove our approach feasible, we intentionally altered the next-state code of the state-machine implementation in a way that the transition from S_7 to S_8 is replaced by a transition from S_7 to S_9, thus conflicting with ψ^3.

Error Scenario. The relevant C code of the implementation is listed in Fig. 4. Whereas the code on the left shows the correct implementation of state *Wait for S_EStopIn1* (S_3), the code on the right erroneously introduces a transition to the state *Reset Error 2* (S_9). We first synthesize hardware observers for ψ^1, ψ^2, and ψ^3. Next, the host application configures the atomic checker unit with the atomic propositions that need to be evaluated, that is:

$$ap_1 : \quad \Theta_{S8} \triangleq (\text{currState} = \texttt{ST\_WAIT\_FOR\_RST2})$$
$$ap_2 : \quad \Theta_{S9} \triangleq (\text{currState} = \texttt{ST\_RST\_ERR2})$$

The (Boolean) verdicts over the atomics are the inputs to the synthesized `ptLTL` observer, i.e., the vector `atomics` of the VHDL entity shown in Fig. 5. The Boolean output `err` is raised to `true` whenever the specification is falsified by the monitor. The sequential process `p_reset` takes care of initialization of the involved registers and the combinatorial process `p_observer_logic` implements the actual observer for ψ^3. We again applied a random input pattern and revealed the erroneous state transition. For example, the sequence $S_1 \succ S_2 \succ S_6 \succ S_7 \succ S_9 \succ S_6$ was shown (by the observer) to be conflicting with specification ψ^3.

4.3 Offline Runtime Verification

To conclude the example, we also applied our offline approach to the *emergency stop* example. We thus synthesized a Java class serving as monitor and used the event logger of the hardware monitor unit to offer state updates δ to the host. Likewise, the host application was also able to reveal the erroneous state transition. However, offline runtime verification requires a host computer to be present, whereas our online approach is a self-contained hardware approach.

```
1 case ST_WAIT_FOR_ESTOPIn2:
2   Ready = true;
3   S_EStopOut = false;
4   Error = false;
5   DiagCode = 0x8004;
6   if (!Activate)
7     currState = ST_IDLE;
8   if(S_EStopIn && !S_AutoReset)
9     currState = ST_WAIT_FOR_RST2;
10  if(S_EStopIn && S_AutoReset)
11    currState = ST_SAFETY_OUTP_EN;
12 break;
```

```
1 case ST_WAIT_FOR_ESTOPIn2:
2   Ready = true;
3   S_EStopOut = false;
4   Error = false;
5   DiagCode = 0x8004;
6   if (!Activate)
7     currState = ST_IDLE;
8   if(S_EStopIn && !S_AutoReset)
9     currState = ST_WAIT_FOR_RST2;
10  if(S_EStopIn && S_AutoReset)
11    currState = ST_RST_ERR2;
12 break;
```

Fig. 4. Emergency stop C implementation; correct(left) and erroneous (right)

5 Related Work

As our approach supports software as well as hardware-based monitoring functionality, we categorize related work into software- and hardware-based approaches.

Software-Based Monitoring. The commercial tool Temporal Rover [8] allows to check future and past time temporal formulae using instrumentation of source code. Basically, the tool is a code generator that supports Java, C, C++, Verilog or VHDL; properties to be checked are embedded in the comments of the source code. The respective property checks are then automatically inserted into the code, compiled, and executed.

Academic tools with automated code instrumentation capabilities are the Java PathExplorer (JPaX) [13], the Monitoring and Checking (MaC) framework [20], and the Requirements Monitoring and Recovery (Rmor) [14] tool. JPaX and MaC facilitate automated instrumentation of Java bytecode; upon execution, they send a sequence of events to an observer. JPaX additionally supports concurrency analysis. Rmor provides a natural textual programming notation for state machines for program monitoring and implements runtime verification for C code.

Hardware-Based Monitoring. Tsai et al. [33] describe a noninterference hardware module based on the MC68000 processor for program execution monitoring and data collection. Events to be monitored, such as function calls, process creation, synchronization, etc. , are predetermined. With the support of a replay controller, test engineers can replay the execution history of the erroneous program in order to determine the origin of the defect. The Dynamic Implementation Verification Architecture (DIVA) exploits runtime verification at intra-processor level [1]. Whenever a DIVA-based microprocessor executes an instruction, the operands and the results are sent to a checker which verifies correctness of the computation; the checker also supports fixing an erroneous operation. A hardware-related tool called BusMop [27] is based on the Monitor Oriented Programming (MOP) framework [6]. In essence, BusMop is a

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity FORMULA_PSI_3 is
5    generic (
6      ATOMICS_LEN : positive := 2;
7      SUBFORMULAS_LEN : positive := 5);
8    port (
9      clk     : in  std_logic;
10     reset   : in  std_logic;
11     atomics : in  std_logic_vector (ATOMICS_LEN-1 downto 0);
12     err     : out std_logic);
13 end FORMULA_PSI_3;
14
15 architecture behaviour of FORMULA_PSI_3 is
16   signal pre_reg, pre_reg_next  :  std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
17   signal now_reg, now_reg_next  :  std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
18   signal atomics_reg            :  std_logic_vector (ATOMICS_LEN-1 downto 0);
19
20 begin
21
22   p_observer_logic : process(pre_reg, now_reg, atomics_reg)
23     variable pre_reg_next_v : std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
24     variable now_reg_next_v : std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
25   begin
26     pre_reg_next_v  :=  pre_reg;
27     now_reg_next_v  :=  now_reg;
28
29     now_reg_next_v(4)  :=  atomics_reg(0);
30     now_reg_next_v(3)  :=  not now_reg_next_v(4) and pre_reg_next_v(4);
31     now_reg_next_v(2)  :=  atomics_reg(1);
32     now_reg_next_v(1)  :=  now_reg_next_v(2) and not pre_reg_next_v(2);
33     now_reg_next_v(0)  :=  not now_reg_next_v(1) or now_reg_next_v(3);
34     pre_reg_next_v     :=  now_reg_next_v;
35
36     pre_reg_next  <= pre_reg_next_v;
37     now_reg_next  <= now_reg_next_v;
38   end process;
39
40   p_reset : process (clk, reset)
41     variable pre_reg_v : std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
42   begin
43     if reset = '1' then
44       pre_reg_v(4)  :=  atomics(0);
45       pre_reg_v(3)  :=  '0';
46       pre_reg_v(2)  :=  atomics(1);
47       pre_reg_v(1)  :=  '0';
48       pre_reg_v(0)  :=  not pre_reg_v(1) or pre_reg_v(3);
49       pre_reg       <= pre_reg_v;
50       now_reg       <= (others => '0');
51       atomics_reg   <= (others => '0');
52     elsif rising_edge(clk) then
53       pre_reg       <= pre_reg_next;
54       now_reg       <= now_reg_next;
55       atomics_reg   <= atomics;
56     end if;
57   end process;
58
59   err <= not now_reg(0);
60
61 end behaviour;
```

Fig. 5. The auto-generated observer VHDL code for ψ^3

hardware-monitoring device which *sniffs* traffic transmitted between COTS embedded components attached to a PCI/PCI-X bus, thereby acting as *advanced* bus guardian. Similar to our approach, the monitor and the system under verification are executed within an FPGA. The specification is translated by the MOP framework into a hardware description, which is then synthesized into a netlist and loaded into dynamically reconfigurable blocks of the FPGA. Whereas BusMop is designed to monitor data transmissions through a PCI interconnection for large-scale embedded systems, our framework monitors embedded software at a fine level of granularity.

The work of Brörkens and Möller [5] is akin to ours in the sense that they also do not rely on code instrumentation to generate event sequences. Their framework, however, targets Java and connects to the bytecode using the Java Debug Interface (JDI) so as to generate sequences of events.

Lu and Forin [25] present a compiler from Property Specification Language (PSL) to Verilog, which translates a subset of PSL assertions about a software program (C in their approach) into hardware execution blocks for an extensible MIPS processor, thus being the first method that allows transparent runtime verification without altering the program under investigation. The synthesized verification unit is generated by a property rewriting algorithm proposed in [32]. Atomic propositions are restricted to allow only a single comparison operator, whereas our approach supports more complex relations among memory values within our hardware unit, thus yielding greater flexibility in the specification.

Observer Synthesis. The idea of generating Java code as observers for ptLTL is due to Havelund and Roşu [15]. A comparable approach based on alternating automata for future time LTL was described by Finkbeiner and Sipma [11].

6 Conclusion and Future Challenges

This paper advocates runtime verification of microcontroller code without code instrumentation. Our method supports runtime checks for ptLTL during execution of the code, thereby evading the problem of errors introduced by translation from a high-level language into binary code. Such errors are likely to go unnoticed by conventional approaches for high-level representations. The framework itself relies on a hardware monitor unit and synthesized observers, thereby making code instrumentation dispensable. The example discussed in this paper is based on randomly generated inputs, which is insufficient in practical applications. Test-case generation for binary code, though orthogonal to the techniques described in this paper, thus remains a topic of interest. For this task, we will further investigate a combination of SAT solving and backward abstract interpretation [30, 4].

Acknowledgement. The work of Thomas Reinbacher and Andreas Steininger has been supported within the FIT-IT project CevTes managed by the Austrian Research Agency FFG under grant 825891. The work of Jörg Brauer and Stefan

Kowalewski has been, in part, supported by the DFG Cluster of Excellence on *Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89 and by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*.

References

1. Austin, T.M.: DIVA: A reliable substrate for deep submicron microarchitecture design. In: MICRO, pp. 196–207. IEEE, Los Alamitos (1999)
2. Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T.: WYSINWYX: What you see is not what you execute. In: VSTTE, Toronto, Canada (2005)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from unstructured programs. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 54–69. Springer, Heidelberg (2011) (to appear)
4. Brauer, J., King, A.: Transfer function synthesis without quantifier elimination. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 97–115. Springer, Heidelberg (2011)
5. Brörkens, M., Möller, M.: Dynamic event generation for runtime checking using the JDI. Electronic Notes in Theoretical Computer Science 70(4), 21–35 (2002)
6. Chen, F., Roşu, G.: MOP: An efficient and generic runtime verification framework. In: OOPSLA, pp. 569–588. ACM, New York (2007)
7. Colin, S., Mariani, L.: Run-Time Verification. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 525–555. Springer, Heidelberg (2005)
8. Drusinsky, D.: The temporal rover and the ATG rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000)
9. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: EMSOFT, pp. 255–264. ACM, New York (2008)
10. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. MIT Press, Cambridge (1990)
11. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Form. Methods Syst. Des. 24, 101–127 (2004)
12. Flexeder, A., Mihaila, B., Petter, M., Seidl, H.: Interprocedural control flow reconstruction. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 188–203. Springer, Heidelberg (2010)
13. Havelund, K., Roşu, G.: An overview of the runtime verification tool Java PathExplorer. Form. Methods Syst. Des. 24(2), 189–215 (2004)
14. Havelund, K.: Runtime verification of C programs. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 7–22. Springer, Heidelberg (2008)
15. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
16. Howe, J.M., King, A.: Logahedra: A new weakly relational domain. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 306–320. Springer, Heidelberg (2009)
17. Kinder, J., Zuleger, F., Veith, H.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 214–228. Springer, Heidelberg (2009)

18. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
19. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: LICS, pp. 383–392. IEEE, Los Alamitos (2002)
20. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: PDPTA, pp. 279–287 (1999)
21. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL, pp. 42–54. ACM, New York (2006)
22. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason 43, 363–446 (2009)
23. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
24. Lindig, C.: Random testing of C calling conventions. In: AADEBUG, pp. 3–12. ACM, New York (2005)
25. Lu, H., Forin, A.: The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Tech. Rep. MSR-TR-2007-99, Microsoft Research (2007)
26. Parr, T.J., Quong, R.W.: ANTLR: a predicated-ll(k) parser generator. Softw. Pract. Exper. 25, 789–810 (1995)
27. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In: Real-Time Systems Symposium, pp. 481–491 (2008)
28. PLCopen: Safety software, technical specification, Part 1: Concepts and function blocks. online (2006)
29. Pnueli, A., Siegel, M.D., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
30. Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., Kowalewski, S.: Test-case generation for embedded binary code using abstract interpretation. In: MEMICS, pp. 151–158 (2010)
31. Reinbacher, T., Horauer, M., Schlich, B., Brauer, J., Scheuer, F.: Model checking assembly code of an industrial knitting machine. In: EM-Com, pp. 97–104. IEEE, Los Alamitos (2009)
32. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Eng. 12(2), 151–197 (2005)
33. Tsai, J.J.P., Fang, K.Y., Chen, H.Y., Bi, Y.: A noninterference monitoring and replay mechanism for real-time software testing and debugging. IEEE Trans. Softw. Eng. 16, 897–916 (1990)

A SAT-Based Approach for the Construction of Reusable Control System Components*

Daniel Côté, Benoî t Fraikin, Marc Frappier, and Richard St-Denis

Département d'informatique
Université de Sherbrooke
Sherbrooke (Québec), J1K 2R1, Canada
{Daniel.R.Cote,Benoit.Fraikin,
Marc.Frappier,Richard.St-Denis}@USherbrooke.ca

Abstract. This paper shows how to take advantage of a SAT-solving approach in the development of safety control software systems for manufacturing plants. In particular, it demonstrates how to construct reusable components which are assembled after instantiation to derive controllers of modular production systems. An experiment has been conducted with Alloy not only to verify properties required by a control theory for complex systems organized hierarchically, but also to synthesize two major parts of a component: observer and supervisor. The former defines its interface while guaranteeing nonblocking hierarchical control. The latter ensures the satisfaction of constraints imposed on its behavior and on the interactions among its subcomponents during system operation. As long as the size of component interfaces is small, SAT-solvers appear useful to build correct reusable components because the formal models that engineers manipulate and analyze are very close to the abstract models of the mathematical theory.

Keywords: Repository of reusable components, component-based software development, hierarchical control, supervisory control theory, verification, synthesis, bounded model checking, SAT-solver, Alloy.

1 Introduction

Control is omnipresent in many industrial critical systems. In order to avoid hazardous operations, the development of controllers cannot be done without the use of rigorous methods because they aim at increasing safety in software solutions. Formal verification techniques have been recognized as essential ingredients of such methods for a long time. However, to be able to analyze software solutions, it is important to understand fundamental rules involved in their construction. This is the reason a control theory, such as the supervisory control theory (SCT) [15], is so promising to get control problems solved properly. A systems-theoretic view of component-based software development (CBSD) that comes with some principles should transcend all technical aspects because these principles promote better software engineering practices based on a sound foundation. Nevertheless, a theory must be accompanied by formal techniques such that together they constitute a standard way for achieving the aforementioned goal.

* The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 52–67, 2011.

Several attempts have been made in this direction since the beginning of the nineties. In particular, symbolic representation techniques have been used with success by some researchers in the framework of SCT for the derivation of optimal supervisors on systems of realistic size. Custom fixed point procedures implemented with binary decision diagrams (BDD) have been developed both in the SCT language-based formulation [3] and SCT state-based formulation [14]. BDD-based symbolic representation techniques have also been used when control specifications are written in a temporal logic [10] or systems are decomposed into subsystems [16]. Furthermore, some academic and commercial tools integrate BDD-based algorithms into their verification and synthesis procedures [20,1]. Now it seems that sizable progress has been achieved through their use in industrial applications (e.g., [9]). Later, efforts have been made to investigate bounded model checking with SAT-solvers to verify the controllability property and deadlock freedom for non-modular and non-hierarchical control [4]. These properties, which appear in almost all control problems, have been encoded, together with the transition functions of automata modeling the plant and control specification, as propositional formulas for checking their satisfiability with respect to the plant and control specification. Automatic generation of nonblocking supervisors has been, however, voluntarily omitted because solving this problem is more difficult using a SAT-solving approach than the verification of the two previous properties. Neither fixed point procedures nor complex encoding schemata are a concern in this paper. The focus is solely on the satisfaction of properties, specified in a *declarative* manner, by SAT-solvers. Compared with the work of Claessen et al. [4], satisfaction of properties is done in a hierarchical manner (not on a flat structure), which reduces the amount of resources required by SAT-solvers. This is due to the fact that control requirements are modularized by subsystems and control problems are solved *locally* at each level of a hierarchy.

In a previous companion paper [6] it has been demonstrated how a systems-theoretic view of CBSD could be advantageous to organize the constituent elements of a complex system into a hierarchy of components in the context of the hierarchical control architecture (HCA) framework of SCT [18]. The emphasis was on formal component properties that are invariant under horizontal composition and vertical composition as well as superposition of control. This allows for their combination in an arbitrary way with an unrestrained number of components and an unrestrained number of abstraction levels No suggestions were, however, made concerning the verification of HCA properties on the results of control problems. It was implicitly assumed that standard, non SAT-based, SCT procedures were used.

In the HCA framework, components are assembled vertically and horizontally, and control becomes explicit when assembling components, since each level of the hierarchy imposes constraints on the level beneath it. With respect to a dynamic closed-loop model, supervisors of adjacent levels are linked by communication channels: a downward communication channel (the command channel) provides control actions in order to enforce constraints on the lower level; and an upward communication channel (the information channel) refines and returns feedback to the upper level. These two channels must be provided with properties that ensure an effective implementation of the high-level supervisor in the lower level. This schema must, however, be transposed into the cyclical scan activity schema usually encountered in programmable logic controllers (PLC) for practical reasons [5].

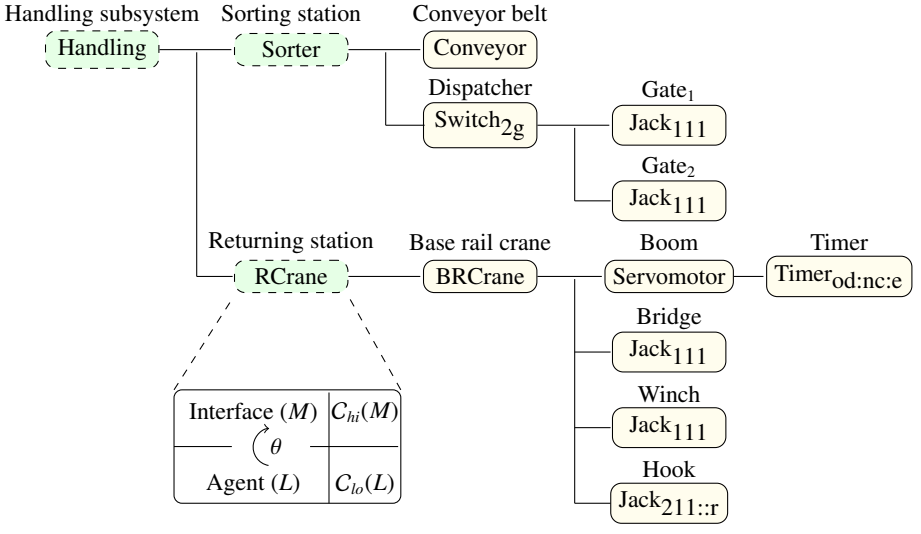


Fig. 1. Hierarchical structure of the handling subsystem

This paper focuses on well-defined procedures for subsystem synthesis and abstraction, starting with elementary components, then composite components. Such components are considered as solutions of control problems solved with the aid of SAT-solvers running relevant models. As long as the procedures for composition, synthesis and abstraction are used, one remains within the HCA framework with all its benefits. In particular this affords building further components by assembly of already abstract components from within the same framework [6]. This opens the way to pyramidal hierarchies of abstract components, potentially compounding the cost savings usually associated with reuse. The dominant rationale behind this approach is to create repositories of reusable components that could pass through a standard certification process that assesses their reliability, which is a crucial issue for industrial critical systems. Overall, the subject is presented as follows. Section 2 makes use of a subsystem of a modular production system (MPS) from FESTO to introduce the main result of the HCA framework and basic elements of an Alloy model. Sections 3 and 4 present the processes and models for the construction of elementary components and composite components with Alloy. Section 5 concludes with some statistics and critical remarks.

2 Illustration of Basic Concepts with an Example

Figure 1 shows the hierarchical structure of a handling subsystem. It is a part of the MPS used as a test bench at GRIL[1] to evaluate controllers derived from various synthesis procedures. It consists of two physically separated stations: the FESTO sorting station [8] and an homemade returning station. The sorting station includes a conveyor

[1] Groupe de recherche en ingénierie du logiciel de l'Université de Sherbrooke.

belt equipped with two gates that can be actuated to dispatch workpieces towards one of three output slides according to some criteria (e.g., color, dimension, material). The returning station has been added to the original MPS in order to implement a continuous manufacturing process. A crane recovers workpieces one by one from the output slides and returns them in a feed magazine located in the first station of the MPS, which makes them available for the next stations. The returning station includes a boom for the motion of a bridge fixed on a two meters square rail linear guide. The bridge permits horizontal movements perpendicular to the rail within a range of 50 centimeters. In a similar way, a winch is attached to the bridge for vertical movements within a range of 40 centimeters. A hook fixed to the winch can pick up and release a workpiece everywhere in this predefined 3D space. The names of these physical modules appear just above the rounding boxes in Figure 1, which represent reusable software components or local software components (dashed outline). These names are also used to designate the corresponding instances of software components. For example, the two gates, the bridge and the winch are all instances of the same reusable component $\texttt{Jack}_{111}$.

The state-space size for this subsystem has been estimated to $7,16 \times 10^8$ states. Even though a nonblocking supervisor could be effectively derived by using some of the BDD-based synthesis procedures mentioned in the introduction, the approach advocated in this paper suggests to apply HCA within the paradigm of CBSD for the following reasons. On the one hand, the notion of interface, which is associated with the upper level in the HCA framework, is an abstraction of the lower level in which some transitions are unobservable. In general, the absence of such a mechanism leads to larger models, particularly in monolithic approaches (e.g., [3,1,10]) or in situations in which the events and states of a transition structure must be renamed when explicitly modeling a given behavior repetitively, because events cannot be shared between internal transition structures (the local coupling property in [14]). In this last case, loss of integrity can occur with respect to the exact state of the corresponding physical module (violation of its correct usage). Even if the notion of interface plays a major role in a modeling method (e.g., [16]), the abstraction mechanism is often limited to a small number of layers by the underlying theory (e.g., two layers in the hierarchical interfaced-based supervisory control theory [13]). On the other hand, the strategy behind HCA promotes a bottom-up approach, but it does not preclude engineers following also a top-down approach. An important point is that engineers can easily trace back control requirements written in a natural language from their formal specifications, particularly when they are expressed in some form of predicates. Furthermore, because requirements are modularized by subsystems, local control problems are smaller and easier to understand. To sum up, applying HCA within the paradigm of CBSD is a less error-prone approach since it makes complexity more manageable and takes advantage of reusable components.

2.1 Overview of the Main Concepts of HCA

The construction of a software component is based on the theorem that appears in the following box (theorem 6 in [18]). At first glance, this theorem provides conditions that must be satisfied at each step of an abstraction process to achieve hierarchical consistency with preservation of nonblockingness.

Let C_{lo} be a standard control structure on L and $\theta : L \to T^*$ be a causal reporter map. Suppose that

$C_{hi}(M) = \theta(C_{lo}(L))$ (control consistency),
$\theta^{-1}(M_m) = L_m$ (consistency of marking),
θ is an observer (observer), and
$\theta_v^{-1} \circ \overline{\kappa}_M \leq \kappa_L \circ \theta^{-1} \circ \overline{\kappa}_M$ (partner-freedom).

Then, for all $E \subseteq M$, $\kappa_M(E)$ is nonblocking $\Leftrightarrow$ $\kappa_L \circ \theta^{-1}(E)$ is nonblocking.

Given two finite sets of events Σ and T, the pair $\langle L, L_m \rangle$ with $L, L_m \subseteq \Sigma^*$ and $L_m \subseteq L = \overline{L}$, where $\overline{L}$ denotes the prefix closure of L [12], is a language model of the lower level (the agent in Figure 1) and $\langle M, M_m \rangle$ with $M, M_m \subseteq T^*$, $M = \theta(L)$ and $M_m \subseteq M = \overline{M}$ is a language model of the upper level (the interface in Figure 1). The map θ and the inverse image map θ^{-1} represent the information channel and command channel between the lower and upper levels, respectively. The term $C_{lo}(L)$ (resp. $C_{hi}(M)$) is the set of controllable sublanguages of L (resp. M) and $\kappa_L(H)$ with $H \subseteq L$ (resp. $\kappa_M(E)$ with $E \subseteq M$) the supremal controllable sublanguage of H (resp. E) w.r.t L (resp. M). Intuitively, $\overline{\kappa_L \circ \theta^{-1}(E)}$ is the behavior of the low-level synthesis of a high-level control specification [18]. Within the context of this theorem, a synthesis procedure that computes $\kappa_M(E)$ while satisfying the nonblocking property ensures that the marked language $\kappa_L \circ \theta^{-1}(E) \subseteq \overline{\kappa_L \circ \theta^{-1}(E)}$ is nonblocking.

The actual form of this theorem represents a serious obstacle to the design and verification of software components. It has been reformulated in a weaker form in [5] for the specific case where both levels use standard control technologies to induce the corresponding standard control structures C_{lo} and C_{hi}. The standard control technology is a partition of the set of events (e.g., Σ) into controllable and uncontrollable events (e.g., Σ_c and Σ_u respectively). An uncontrollable event cannot be disabled by a supervisor. Let $L_{voc} := \{\epsilon\} \cup \omega^{-1}(T)$, where $\omega : L \to T$, called the tail map of θ, is defined as follows ($s \in \Sigma^*$, $\sigma \in \Sigma$ and $\tau \in T$):

$$\omega(s\sigma) := \begin{cases} \tau, & \text{if } \theta(s\sigma) = \theta(s)\tau; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The set L_{voc} is the set of vocal strings, which are the strings of L that cause the generation of an event through θ [18]. The map θ can then be expressed as follows:

$$\theta(\epsilon) = \epsilon$$
$$\theta(s\sigma) = \begin{cases} \theta(s)\omega(s\sigma), & \text{if } \omega(s\sigma) \text{ is defined;} \\ \theta(s) & \text{otherwise.} \end{cases}$$

Let $X_\tau := \{s\sigma \in L_{voc} \mid \omega(s\sigma) = \tau\}$ and $T_\theta := \{\tau \in T \mid X_\tau \neq \emptyset\}$. With respect to the general theorem, the control consistency and partner-freedom properties have been eliminated as described in the following box.

Let $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$ be the standard control technology for the agent, $T = T_c \dot{\cup} T_u$ with $T_c = \{\tau \in T_\theta \mid X_\tau \subseteq \Sigma^* \Sigma_c\}$ and $T_u = T_\theta - T_c$ be the standard control technology for the interface, and $\theta : L \to T^*$ be a causal reporter map. Suppose that

$\theta^{-1}(M_m) = L_m$ (consistency of marking) and
θ is an observer (observer).

Then, for all $E \subseteq M$, $\kappa_M(E)$ is nonblocking $\Leftrightarrow$ $\theta^{-1}(\kappa_M(E))$ is nonblocking.

This result is due to the standard control technology peculiar to the upper level (the choice of T_c). Indeed, it implies control coincidence ($\theta^{-1}(C_{hi}(M)) \subseteq C_{lo}(L)$), which in turn implies a weak form of control consistency ($C_{hi}(M) \subseteq \theta(C_{lo}(L))$) and partner freedom [5]. It should be noted that $\kappa_M(E)$ is used instead of E (this corresponds to an appropriate choice of E with respect to the original theorem) and κ_M is idempotent. If $\kappa_M(E)$ is nonblocking, then the corresponding behavior in the lower level is nonblocking which means that deadlock and livelock cannot happen.

Compared with strict control consistency, this result gives access to fewer control options in C_{lo} by the upper level. Thus, control becomes coarser as the hierarchy of abstraction builds up. Nevertheless, it often reveals adequate for practical use as it embodies the usual trade-off made to obtain coarser-grained interfaces (i.e., simpler abstractions) in order to encapsulate complexity.

2.2 Overview of ALLOY and Its Use in the Design of Components

ALLOY is a symbolic model checker [11]. Its modeling language is first-order logic with relations as the only type of terms. Basic sets and relations are defined using *signatures*, a construct similar to classes in object-oriented programming languages, which supports inheritance. ALLOY uses SAT-solvers to verify the satisfiability of axioms defined in a model and find counterexamples for properties (theorems) that should be deduced from these axioms. An ALLOY specification consists of signatures, noted `sig`, which basically define sets and relations. Constraints, noted `fact` or attached to a signature as appended facts (lines 9–14 in the following specification), are formulas that condition the values of sets and relations. The declaration `sig X {r : Y -> Z}` declares a set `X` and a ternary relation `r` which is a subset of the Cartesian product $X \times Y \times Z$.

In the sequel ALLOY is used to derive supervisors and verify consistency of marking and the observer property. Since the choice of T_c by engineers must conform to the definition of the standard control technology of an interface, it must be checked by ALLOY. These operations are performed on automata specified as follows in the ALLOY modeling language.

```
1  abstract sig State {}
2  abstract sig Event {}
3  abstract sig Automaton
4  {  states: set State,
5     events: set Event,
6     initialState: lone State,
7     finalStates: set State,
8     transition: set State -> Event -> State
9  }{
10   transition.dom + transition.ran in states
11   transition.mid in events
12   initialState in states
13   finalStates in states
14  }
15
16 fun getReachableStates[a: Automaton, s: set State] :
17     set State
```

```
18 { s.*(a.transitionsOn[a.events]) }
19
20 fun getCoreachableStates[a: Automaton, s: set State] :
21     set State
22 { s.*(~(a.transitionsOn[a.events])) }
23
24 pred isTrim[a: Automaton]
25 {
26  let S = getReachableStates[a, a.initialState] &
27          getCoreachableStates[a, a.finalStates]
28        | a.transition = a.transition & (S -> Event -> S)
29          and a.finalStates = a.finalStates & S
30 }
```

In this model the function `transitionsOn` returns the pairs of states of transitions because the events are irrelevant for determining reachable and coreachable states. Operator "`*`" is the reflexive transitive closure, "`~`" is the inverse operator and "`&`" is the intersection. The first two apply on a relation. The predicate `isTrim` is satisfied when the automaton `a` has only reachable and coreachable states, in particular the system behavior represented by the automaton is nonblocking.

3 Construction of Elementary Components with ALLOY

An elementary component is generally associated with a hardware device. It is created in an ad hoc manner from a raw model that represents the device in terms of a state space defined from the value domains of sensors and actuators, and all the events generated by the device. The events and states are observable by a control device such as a PLC. Several assumptions must be made on the raw model in order to obtain a suitable model that depicts the real behavior of the device. This step allows for elimination of instability problems such as inertial effects due to a chaotic usage of a command. These assumptions are proven to be acceptable with respect to statistical tests about meticulous experiments on the device. The suitable model is then refined into more expressive models by relabeling events and states. These expressive models aim to provide interfaces with the component.

Figure 2 shows the raw model of a typical pneumatic jack (or cylinder) with five self-locking valves. Pneumatic jacks come in various forms. Their main features are the number of valves to regulate the exhaust air flow (e.g., three valves for one stable position or five valves for two stable positions), the type of valves (e.g., self-locking valve) and the presence of sensors to detect the end of travel positions. The lack of sensors at either end of travel positions requires a timer to assess complete extension or complete retraction. In this model, Xi and Yi denote sensor inputs and actuator commands,[2] respectively, and it is assumed that the end of travel positions are both detected by sensors. The transitions represented by dashed edges have been identified as nonessential following experiments on its working. Eliminating these transitions yields a state transition system, which models the device behavior in a normal mode of operation (the

[2] More precisely, Xi and $\overline{\text{X}i}$ denote the values of a binary sensor while $\uparrow$Xi and $\downarrow$Xi denote a signal rising edge and a signal falling edge, respectively.

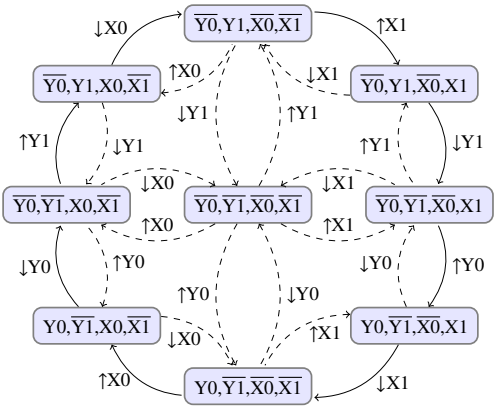


Fig. 2. Raw model of $\texttt{Jack}_{211}\texttt{::r}$

automaton at the left of Figure 3), and control actions, which disable the superfluous controllable transitions (omitted due to space limitation). It takes 589 ms to check the predicate `isTrim` for this automaton with Alloy (and MiniSat) running on a 2,5 GHz Intel dual-core with 4 Gb of memory. As a matter of fact, the automaton at the left of Figure 3 is a Mealy machine that also defines the causal reporter map θ. Applying this map gives the interface at the right of Figure 3. Essentially, it permits to extend and then retract the cylinder repetitively. Indeed, several interfaces are possible. They may differ in the number of commands they provide and their initial state. Such an abstraction reduces the number of states to consider at the next layer. Even if the reduction ratio is small, the combined effect on several layers may be considerable.

The following part of an Alloy model contains a representation of the map θ in a relational fashion. It has one argument, an input/output string in the sense of a Mealy machine with the input string over the input alphabet Σ (e.g., $\{\uparrow X0, \downarrow X0, \uparrow X1, \downarrow X1, \uparrow Y0, \downarrow Y0, \uparrow Y1, \downarrow Y1\}$) and the output string over the output alphabet T (e.g., $\{$*extend*, *eoe*, *retract*, *eor*$\}$). It returns the corresponding output string over T, which is defined as an Alloy sequence (`seq`) of symbols (i.e., a relation of $\mathbb{Z} \times T$). The local variable `sq` defines the set of pairs (i, o) of a sequence that corresponds to the output string of the input/output string. The function `inds` returns the indexes associated with a sequence. The fact at line 7 forces θ to be total.

```
1 fun theta[s: IO/String] : O/String
2 { let sq = { i: Int, o: T | i in inds[s.sequence] and
3                             o = (s.sequence)[i].oLabel }
4           | { t: O/String | t.sequence = sq }
5 }
6 fact
7 { all s: IO/String | some t: O/String | theta[s] = t }
```

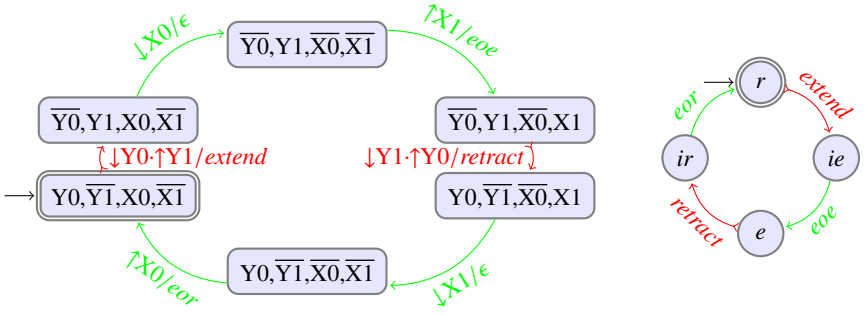


Fig. 3. Agent and an interface of $\texttt{Jack}_{211}\texttt{::r}$

3.1 Checking the Condition for T_c

The choice of the controllable events T_c by engineers must fulfill the condition prescribed by the weak form of the theorem introduced in Section 2.1 (i.e., $T_c = \{\tau \in T_\theta \mid X_\tau \subseteq \Sigma^* \Sigma_c\}$). This condition is expressed as follows in Alloy.

```
pred ISCT
{ all t : IO/String | not t.lastEvent.isSilent and t.isPrefix
          implies all s : IO/String | theta[s] = t implies
                  t.lastEventCntl implies s.lastEventCntl
}
```

Given a string t generated by the interface (`t.isPrefix`) for which its last event is non-silent (`not t.lastEvent.isSilent`), then for all string s such that $\theta(s) = t$ ($s \in L_{voc}$ necessarily holds), if the last event of t is controllable (`t.lastEventCntl`) then the last event of s must be controllable (`s.lastEventCntl`). It takes 639 ms to check that the component $\texttt{Jack}_{211}\texttt{::r}$ satisfies this predicate.

3.2 Checking the Consistency of Marking

The consistency of marking property, $\theta^{-1}(M_m) = L_m$, is formulated as follows in Alloy.

```
pred CM
{ all s : IO/String | s.isPrefix implies
          let q = lastVisitedState[s] | theta[s].isAccepted
          iff some IOAutomaton.finalStates & eClosure[q]
}
pred isAccepted[x: String]
{ x.isPrefix
  some (last[x.visitedStates] & x.wrtAutomaton.finalStates)
}
pred isPrefix[x: String]
{ #inds[x.visitedStates] = #inds[x.sequence]+1 }
```

The auxiliary predicates `isAccepted` and `isPrefix` are self-understandable. They hold when the last visited state from the initial state is an accepting state and when

the number of visited states from the initial state is equal the length of the string plus one, respectively. The function `eClosure` (for ϵ-closure) is defined from the function `GetReachableStates`. The predicate `CM` needs explanation. If $s \in L$ (`s.isPrefix`) then its image $\theta(s) \in M_m$ (`theta[s].isAccepted`) if and only if there is a final state in the ϵ-closure(q) with respect to the output alphabet, where q is the last state visited by s (i.e., $(\exists u \in \Sigma^*) su \in L_m \wedge \theta(su) = t$). Consistency of marking can be checked from a given agent and a given interface by using the following command:

```
1 check { CM } for 1 but length seq, 1 O/String, 1 IO/String
```

In this command, the symbol `length` must be replaced by an integer constant equal to the maximum length of a sequence of states that corresponds to a string (i.e., `s` $\in$ `IO/String`) in order to be sure to deny the predicate, if such a string exists. This bound can be easily determined. It is the number of states of the agent plus one. For the component $\texttt{Jack}_{211}\texttt{::r}$, `length` is set to 7 and Alloy verifies the predicate in 704 ms.

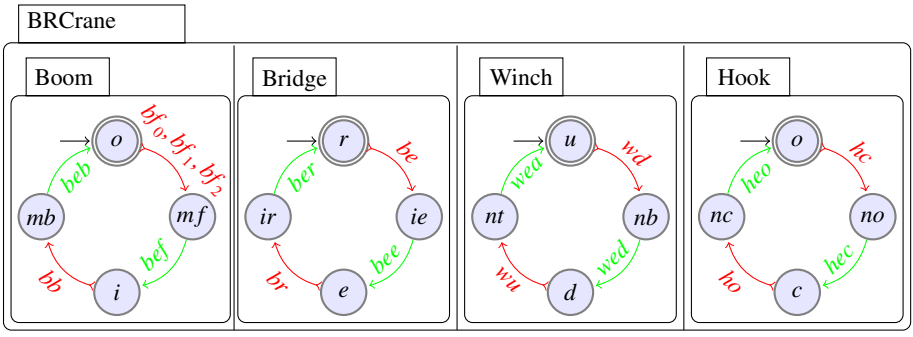
3.3 Checking the Observer Property

A causal reporter map involves information hiding and relabeling. During the construction of a component, engineers guess θ based on the wanted interface for the component while trying to satisfy the observer property. This property, which is closely related to the concept of observational equivalence, can be characterized as follows (application of lemma 2.1 in [19], see also [18]): θ is an observer *iff* $(\forall s \in L)(\forall \tau \in T)\theta(s)\tau \in M \implies (\exists u \in \Sigma^+) su \in L \wedge \theta(su) = \theta(s)\tau$.

```
1 pred thetaIsAnObserver
2 {
3  all s : IO/String | all t : O/String | all o : T |
4           s.isPrefix and not o.isSilent and t.isSymbol[o] and
5           (theta[s]).followedBy[t].isPrefix implies
6           some q : IOAutomaton.states
7                  | let p = lastVisitedState[s]
8                  | q in eClosure[p] and o in outputFrom[q]
9 }
```

The term `s.isPrefix` at line 4 means that $s \in L$ and `not o.isSilent` at the same line means that $\tau \in T$. The term `t.isSymbol[o]` is similar to a cast operation because of the distinction between a symbol of T and a string of length one over T in the Alloy model. The term `(theta[s]).followedBy[t].isPrefix` at line 6 is the translation of the condition $\theta(s)\tau \in M$ in Alloy. The formula at lines 7–8 corresponds to conclusion $(\exists u \in \Sigma^+) su \in L \wedge \theta(su) = \theta(s)\tau$, but in terms of the automaton representation of u. It takes 1 864 ms to check that θ of the component $\texttt{Jack}_{211}\texttt{::r}$ satisfies this predicate.

If a given θ does not satisfy the predicate, then a counterexample is generated by Alloy. It can be used to identify a faulty transition from which a new θ can be proposed. Satisfying consistency of marking and the observer property together represents an iterative procedure in which human intervention is required in order to obtain a useful interface. Such a procedure can be implemented by using the *Kodkod* API [17].



o:	boom at the output position	*r*:	bridge retracted	*u*:	winch upstairs	*o*:	hook open
mf:	boom moving forward	*ie*:	bridge in extension	*nb*:	winch near bottom	*no*:	hook not open
i:	boom at the input position	*e*:	bridge extended	*d*:	winch downstairs	*c*:	hook closed
mb:	boom moving backward	*ir*:	bridge in retraction	*nt*:	winch near top	*nc*:	hook not closed
bf_i:	move forward	*be*:	extend	*wd*:	go down	*hc*:	close
bef:	end of moving forward	*bee*:	end of extension	*wed*:	end of descent	*hec*:	end of closing
bb:	move backward	*br*:	retract	*wu*:	go up	*ho*:	open
beb:	end of moving backward	*ber*:	end of retraction	*wea*:	end of ascent	*heo*:	end of opening

Fig. 4. Assembly diagram of the composite component `BRCrane`

4 Construction of Composite Components with ALLOY

Composite components may also be included in a repository of reusable components. Usually, such components are common in transportation (e.g., crane, conveyor) and processing (e.g., drill, measurement instrument) of workpieces. Contrary to elementary components, they are built by following a systematic design process, in particular their control part can be automatically derived with respect to a control specification. The construction of a composite component is illustrated with a typical crane, named `BRCrane`, introduced in Section 2. It is created from one composite component (a boom) and three elementary components (a bridge, a winch and a hook). As shown in Figure 1, the boom is an instance of `Servomotor`, the bridge and the winch are instances of $\texttt{Jack}_{111}$, and the hook is an instance of $\texttt{Jack}_{211}\texttt{::r}$. Figure 4 gives the assembly diagram of this component. It includes the interfaces of the basic components after instantiation.

The control specification associated with this component includes 19 constraints classified into three groups: *strict shuffling of command pair*, *equipment safety* and *functionality*. They are expressed in propositional logic formulas which define a set of forbidden states. For example:

- $Boom \in \{mf, mb\} \wedge Bridge \in \{ie, ir\}$ is a strict shuffling of command pair constraint. When negated this formula means that the boom and the bridge cannot move simultaneously.
- $Boom = o \wedge Winch = d$ is an equipment safety constraint. When negated, it forbids the winch to be pulled down when the boom is just above the feed magazine.
- $Boom = mf \wedge Hook = nc$ is a functionality constraint. When negated, it states that the command to open the hook cannot be launched when the boom is moving

forward to reach one of the three output slides, because it is assumed that it is already open and it will be closed.

This description seems to give the impression that this composite component can only be used in the specific context of the MPS. This is not absolutely true because this crane can be seen as a transportation device that picks up a workpiece from three possible input positions and move it to a unique output position, vertically higher and horizontally distant from the input positions, where it is put down. Furthermore, it can be useful to create other components that model similar cranes, but with different, even more permissive, behaviors.

The free behavior of the crane is obtained by the shuffle product of the four automata that appear in Figure 4. This yields to an automaton with 256 states (denoted by `aut` in the following ALLOY model) in which some events are uncontrollable (`uEvents`). At this stage, the maximally permissive and nonblocking supervisor (`sup`), that forbids the crane reaching any state that belongs to the set of forbidden (or bad) states (`bStates`), must be derived. The predicate `goodStates` specifies the set of states of an acceptable solution, that is, a solution in which there is no sequence of uncontrollable transitions that leads to a bad state (controllability property) and the good states are coreachable to the set of final states or equivalently any task can be completed (nonblocking property).

```
1  pred goodStates[a: Automaton,
2                  uEvents: set Event,
3                  bStates, gStates: set State]
4  {
5   let gT = gStates <: (a.transitionsOn[Event]) :> gStates,
6       uT = (a.transitionsOn[uEvents]) |
7       {
8        gStates in (a.initialState).*gT
9        no (*uT.bStates & gStates)
10       all q : gStates | some (q.*gT & a.finalStates)
11       gStates.uT in gStates
12      }
13 }
```

The term `gT` defined at line 5 by using the domain restriction (`<:`) and range restriction (`:>`) operators represents the set of transitions between good states (`gStates`). The term `uT` defined at line 6 denotes the set of uncontrollable transitions. The good states are reachable from the initial state (line 8), coreachable to the set of final states (line 10) and closed under uncontrollable transitions (line 11). Furthermore, no chain of uncontrollable transitions leads to a forbidden state from a good state (line 9).

Unfortunately, nothing guarantees that the instance found by ALLOY is effectively the supremal solution. Nevertheless, this has been the case for all the supervisors derived for the MPS and other systems until now. A safe procedure to synthesize the supremal solution consists in defining a predicate with the following formula.

```
1  sup.isSupervisor[aut, uEvents, bStates]
2  all x : Supervisor | x.isSupervisor[aut, uEvents, bStates]
3          => x.states in sup.states
```

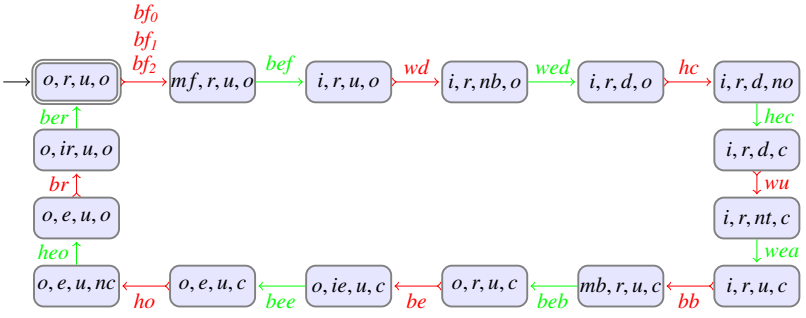
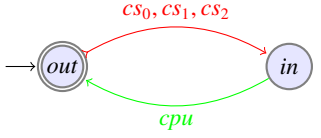


Fig. 5. `BRCrane` agent under supervision



out: crane at the output position
in: crane at the input position
cs_i: shift at position i ($0 \leq i \leq 2$)
cpu: pick-up

Fig. 6. `BRCrane` interface

If x is an acceptable solution of the same control problem, the states of x must be included in the set of states of a given solution found by Alloy in the previous iteration. At this point, two cases are possible. First, there is no counterexample, which means that *sup* is the supremal solution. Second, there is a counterexample. Adding the states of x that are not included in *sup* to the set of states of *sup* is another acceptable, but greater solution, which must be used for the next iteration. Such a procedure can be implemented by using again the *Kodkod* API.

Deriving the maximally permissive and nonblocking supervisor by using a SAT-solver is unusual, even though it is equivalent to some fixed-point procedures defined on languages [12] or synthesis algorithms working on automata [2]. It takes 13 480 ms to derive a supervisor for the crane and 22 770 ms to check that it is the least restrictive. It is given in Figure 5. Since the number of atoms is limited to about 256 in Alloy when there is a quaternary relation in a model, the set of forbidden states (231 states) has been replaced by a unique representative state to get round this limitation.

It should be noted that the two properties involved in the weak form of the theorem introduced in Section 2.1 are preserved when components are aggregated horizontally or vertically without adding supplementary control, and when supplementary control is superposed to the control of the interface of a component [6]. Indeed, the boom, the bridge, the winch and the hook has been aggregated horizontally, then control has been added to supervise the agent of the crane, not the interfaces of the basic components individually. Eventually, control could be added to the interface of the crane. The latter could be considered as the agent of a new component in which control is added to constrain even more the behavior of the crane.

Finally, the interface of the crane (see Figure 6) is obtained by a causal reporter map that satisfies the condition on T_c, consistency of marking and observer property. It takes 3 542 ms, 3 800 ms and 16 100 ms to check the corresponding predicates with Alloy.

5 Conclusion

The work reported in this paper has focused on two main points. First, it shows that a variant of the original HCA framework combined with CBSD is advantageous to construct a repository of reusable components and use it in the development of control software systems for modular production systems. Second, it reveals how a SAT-solving approach is applicable to verify properties and synthesize parts of small components, since the Alloy models have been obtained by a direct translation of the theory. This work has resulted in a repository presently containing 48 elementary components and 17 composite components. It has been exploited with success in the systematic derivation of a S7-SCL program by a naive (not yet optimized) code generation procedure from glue code of reusable components after instantiation. The program has 6 631 lines of source code (with comments) running on a *Siemens* PLC, model *315PN/DP*, connected to a *Profibus* network. The size of the object code is bigger than the one obtained from an ad hoc solution programmed in the S7-GRAPH language (36 580 bytes versus 26 718 bytes).

The MPS from FESTO used in the case study, with all its pneumatic, mechanical and electrical apparatuses similar to those encountered in the manufacturing industry, and the data in Table 1 demonstrate the relevance to industrial application. To the best of our knowledge, it is the first time that the HCA framework has been adapted in order to solve a control problem of substantial size in the context of CBSD and SAT-solving. This work also demonstrates how the underlying hierarchical design method combined with CBSD is scalable. Of course, using a SAT-solving approach to built a repository of reusable components entails drawbacks mainly due to limitations imposed by Alloy, in particular the maximum number of allocated tokens, which impacts on the maximal size (in terms of number of states) of automata. Among the 17 composite components, two have not been constructed by the proposed approach, likewise for the *distribution subsystem*, *testing subsystem* and *processing subsystem*. Different encoding schemata of the interfaces of their subcomponents and automaton used for synchronization should be examined to settle the problem. The run times observed with Alloy are significantly higher than those gathered during the use of *Supremica* [1] (less than 350 ms for each synthesis of a supervisor) and execution of a *C++* implementation of the algorithm developed by Wong and Wonham [19] for computing an observer (less than one second for each observer), which is an adaptation of the Fernandez algorithm [7]. However, no formal proof of correctness exists for the aforementioned programs. Since they are complex such proofs represent a particularly difficult task. Checking the correctness of an Alloy specification with respect to a control theory is tractable, because it is essentially a translation of mathematical conditions of the theory into Alloy code. Moreover, Alloy makes it easier to maintain these specifications (only 400 lines in this work) to take into account the evolution of a control theory. The conclusion of this comparative

Table 1. Statistics about the development of the control software system for the MPS

	Distribution subsystem	Testing subsystem	Processing subsystem	Handling subsystem	MPS
Estimated size of the state space	$1,10 \times 10^4$	$6,91 \times 10^4$	$7,13 \times 10^{13}$	$7,16 \times 10^8$	$1,5 \times 10^{32}$
Number of components	5	6	19	13	44
Number of memories	1	2	10	7	22
Number of layers	3	3	5	5	6
Control specification	22	22	15	10	4
Control specification (reuse)	10	14	79	50	222
Total	32	36	94	60	226
Size of source code (LOC)	674	785	2 504	1 777	6 631
Size of object code (bytes)	3 944	3 964	13 454	9 948	36 580

study confirms that the proposed approach can deal with interesting problem encountered in industry considering the current state-of-the-art of SAT solvers. Despite the underlying difficulties, this type of research (adopting the SAT-solving paradigm to model a control theory in the practitioner's CBSD perspective in order to achieve scalability and correctness) is promising.

References

1. Åkesson, K., Fabian, M., Flordal, H., Malik, R.: Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In: Proceedings of 8th International Workshop on Discrete Event Systems, pp. 384–385 (2006)
2. Barbeau, M., Kabanza, F., St-Denis, R.: An efficient algorithm for controller synthesis under full observation. Journal of Algorithms 25, 144–161 (1997)
3. Balemi, S., Hoffmann, G.J., Gyugyi, P., Wong-Toi, H., Franklin, G.F.: Supervisory control of a rapid thermal multiprocessor. IEEE Transactions on Automatic Control 38, 1040–1059 (1993)
4. Claessen, K., Een, N., Sheeran, M., Sörensson, N., Voronov, A., Åkesson, K.: SAT-solving in practice, with a tutorial example from supervisory control. Discrete Event Dynamic Systems: Theory and Applications 19, 495–524 (2009)
5. Côté, D.: Conception par composantes de contrôleurs d'usines modulaires utilisant la théorie du contrôle supervisé. Ph.D. thesis, Département d'informatique, Université de Sherbrooke, submitted (2011)
6. Côté, D., Embe Jiague, M., St-Denis, R.: Systems-theoretic view of component-based software development. In: Pre-proceedings of 7th International Workshop on Formal Aspects of Component Software, pp. 65–82 (2010) (to appear in Lecture Notes in Computer Science)
7. Fernandez, J.-C.: An implementation of an efficient algorithm for bisimulation equivalence. Science of Computer Programming 13, 219–236 (1990)
8. FESTO: Sorting Station—Modular Production System. Festo Didactic GmbH & Co., Denkendorf (1998)
9. Gebremichael, B., Vaandrager, F.: Control synthesis for a smart card personalization system using symbolic model checking. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 189–203. Springer, Heidelberg (2004)

10. Gromyko, A., Pistore, M., Traverso, P.: A tool for controller synthesis via symbolic model checking. In: Proceedings of 8th International Workshop on Discrete Event Systems, pp. 475–476 (2006)
11. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2006)
12. Kumar, R., Garg, V.K.: Modeling and Control of Logical Discrete Event Systems. Kluwer Academic Publishers, Boston (1995)
13. Leduc, R.J., Lawford, M., Wonham, W.M.: Hierarchical interface-based supervisory control—part II: parallel case. IEEE Transactions on Automatic Control 50, 1336–1348 (2005)
14. Ma, C., Wonham, W.M.: Nonblocking Supervisory Control of State Tree Structures. Lecture Notes in Control and Information Sciences, vol. 317. Springer, Heidelberg (2005)
15. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE 77, 81–98 (1989)
16. Song, R., Leduc, R.J.: Symbolic synthesis and verification of hierarchical interface-based supervisory control. In: Proceedings of 8th International Workshop on Discrete Event Systems, pp. 419–426 (2006)
17. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
18. Wong, K.C., Wonham, W.M.: Hierarchical control of discrete-event systems. Discrete Event Dynamic Systems: Theory and Applications 6, 241–273 (1996)
19. Wong, K.C., Wonham, W.M.: On the computation of observers in discrete-event systems. Discrete Event Dynamic Systems: Theory and Applications 14, 55–107 (1996)
20. Zhang, Z., Wonham, W.M.: STCT: an efficient algorithm for supervisory control design. In: Caillaud, B., Darondeau, P., Lavagno, L., Xie, X. (eds.) Synthesis and Control of Discrete Event Systems, pp. 77–102. Kluwer Academic Publishers, The Netherlands (2002)

Formal Safety Analysis in Industrial Practice

Ilyas Daskaya[1], Michaela Huhn[2], and Stefan Milius[1]

[1] Institut für Theoretische Informatik, Technische Universität Braunschweig
Braunschweig, Germany
{I.Daskaya,S.Milius}@tu-braunschweig.de
[2] Department of Informatics, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
Michaela.Huhn@tu-clausthal.de

Abstract. We report on a comparative study on formal verification of two level crossing controllers that were developed using SCADE by a rail automation manufacturer. Deductive Cause-Consequence Analysis of Ortmeier *et al.* is applied for formal safety analysis and in addition, safety requirements are proven. Even with these medium size industrial case studies we observed intense complexity problems that could not be overcome by employing different heuristics like abstraction and compositional verification. In particular, we failed to prove a crucial liveness property within the SCADE framework stating that an unsafe state will not be persistent. We finally succeeded to prove this property by combining abstraction and model transformation from SCADE to UPPAAL timed automata. In addition, we found that the modeling style has a significant impact on the complexity of the verification task.

Keywords: model-based development, SCADE, Deductive Cause-Consequence Analysis.

1 Introduction

A key issue in the development of safety-critical systems is safety analysis, i. e., a thorough analysis how components may fail and cause a system hazard. From the safety analysis, the safety requirements for components are derived and the design is proven correct w.r.t. functional safety in the verification and validation phase.

For developing software for safety-critical systems, formal methods are considered an adequate means because they provide correctness results at a level of strict mathematical rigor. Standards like the IEC 61508 part 3 [15] and its railway-specific derivative CENELEC 50128 [6] highly recommend them for the software development according to higher safety integrity levels (SIL). Nevertheless, seamless usage of formal methods is still a future challenge for industries. Frequently heard arguments trying to explain industries' indecision towards the proliferation of formal methods are a lack of integration and coverage in the development process and poor scalability for larger designs.

In this paper we report on our experiences on a seamless, tool supported, formal approach that covers design, safety analysis and formal verification: We compare two medium-sized design variants of a level crossing controller. Both were developed using

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 68–84, 2011.

the SCADE Suite[1] by a manufacturer for rail automation equipment. The two models are different in their modeling style, namely one is state-based and built based on safe state machines [4], whereas the other one is strictly data-flow oriented. Both implement the same functionality and have exactly the same interfaces, but the models were originally thought as a comparative basis to evaluate model qualities relevant for application developers such as understandability, maintainability etc. This was a purely design-oriented comparison that we have expanded to safety analysis and formal verification.

So the starting point for our investigation is the question whether the modeling style has an impact on formal safety analysis and verification. By using the built-in state-of-the-art SAT-based model checker of SCADE we also strive for a good showcase for seamless formal support of the safety process. A precondition for an authentic showcase is to take the original design models without tailoring them beforehand to the needs of some particular formal analysis technique at hand.

We apply *Deductive Cause-Consequence Analysis* (DCCA) by Ortmeier et al. [19] as a formal generalization of the well accepted safety analysis techniques FMEA (*Failure Mode and Effect Analysis*) and FTA (*Fault Tree Analysis*). Moreover, we are able to formally verify a number of safety requirements, i.e. functional correctness properties imposed by the safety analysis. However, we have to cope with severe complexity problems even with these medium sized designs that prevent us from completing both, the DCCA and the verification of safety requirements. We try several abstraction techniques to reduce complexity of the verification problems within the SCADE Design Verifier, but without success. We illustrate our attempts on data abstraction, cone of influence, and symmetry reduction as well as decomposition using the example of a liveness property that arises from the safety requirements. Finally, we transform the state-based model into UPPAAL timed automata[2] and succeed with the verification of that liveness property easily. What we technically realize as a model transformation between two formal frameworks, namely SCADE and UPPAAL, is methodically an *abstraction of time*: In the SCADE model time is handled in multiple steps in each of which an external timer is compared to internal variables modeling timeouts. In contrast, the real-time zones of UPPAAL only take *one* transition to the next relevant point in time. Hence the resulting state space is significantly smaller.

Overall, the state-based model of the level crossing control can be considered slightly better suited for formal analysis. However, to be able to generalize this result, a clear and *application-oriented* notion of model elements with major impact on the verification complexity has to be developed.

The paper is structured as follows: In Sec. 2 we recall safety analysis using FMEA and DCCA as a formal approach to it and SCADE suite as a model-based development environment featuring formal verification. Liveness analysis, a number of heuristics we have tried in order to deal with the intrinsic complexity problems, and, notably, the transformation of SCADE models to UPPAAL timed automata are described in Sec. 3. In Sec. 4, safety analysis and verification results of the two model variants of a level crossing controller are presented in detail. Sec. 5 concludes with lessons learned.

[1] SCADE is a product of Esterel Technologies, see `www.esterel-technologies.com`.

[2] UPPAAL is an integrated tool for modeling and verification of real-time systems, see `www.uppaal.com`.

2 Safety Analysis and DCCA

The development of safety-critical systems and their software is regulated by standards. In the railway domain the CENELEC standards EN 50126, 50128, and 50129 [9,6,10] apply. Since software is intangible, it is commonly agreed that system failures caused by software malfunction stem from systematic errors that are introduced during the software development and are not recognized in the safety process. Consequently, software safety engineering aims at (1) a complete and consistent specification of functional and safety aspects for a software component, (2) the correct implementation of the specification and (3) providing evidence that the safety objectives are met.

2.1 Safety Analysis

In the architectural design phase, the intended functionality is modularized. For safety-critical systems, a *hazard analysis* is performed to identify potential failures, hazards, and the causal chains between them. Classical inductive methods for hazard analysis are *Failure Mode and Effect Analysis* (FMEA) and its extension *Failure Mode, Effects and Criticality Analysis* as standardized in IEC 60812 [14]: FMEA classifies failures according to the severity of their effects, the occurrence frequency, and the detection rate. Starting from a definition of the system and its boundaries, a functional viewpoint is taken and each subfunction is analyzed with respect to potential fault modes. For functionality implemented in software, fault identification is supported by generic fault types like omission, commission, untimely reaction and value fault [18]. Local effects can be directly deduced from the component faults. In order to determine the system level effects, fault propagation is analyzed based on the functional architecture by using a formal deduction method (see Section 2.4). A fault is called *critical* if it has severe effects and an unacceptably high occurrence frequency. Safety measures are taken to eliminate the faults or at least mitigate the effects, to decrease their occurrence probability, or to actively detect them. Findings and actions to be taken are usually summarized in an FMEA table. Often a Fault Tree Analysis (FTA) is conducted to complement FMEA. FTA proceeds deductively by starting from potential hazards and then investigates which combination of faults or operational modes in the components may cause them.

FMEA is a structured but semi-formal technique. In an iterative design process, an FMEA is conducted and refined with each iteration cycle or change in the design or operational constraints. As FMEA is an inductive method, common cause faults and their effects are analyzed in a separate step.

2.2 Safety-Oriented Software Design

For the development of safety-related software in the rail domain, the standard CENELEC 50128 [6] prescribes the activities for design, validation, and verification with their input and output artifacts. As faults due to software are traced back to systematic development errors, the standard recommends measures that are considered appropriate to avoid such errors or to reveal them in a validation or verification step. Hence,

development activities have to be performed with an adequate degree of rigor such that functional correctness and fulfillment of safety requirements can be proven with substantial evidence.

The SCADE tool suite (Safety-Critical Application Development Environment) is a model-based development framework for safety-critical software that supports certification due to EN 50128 up to SIL 3 and 4. The SCADE modeling language is a synchronous and dataflow-oriented language based on LUSTRE [12], and was extended by safe state machines [4]. SCADE provides certified automated code generation and immediate simulation of the model. It supports testing, in particular model coverage is recorded, and the SCADE Design Verifier (SCADE DV) allows for SAT-based formal verification[3].

Here, SCADE was chosen as design framework for the level crossing control (LC) from our industrial partner. The reason was the seamless design flow from the requirements via the model executing on a hardware abstraction layer (HAL) to the C code generated for the real target processor. Back then, validation, verification and safety assessment were performed with traditional methods. With a broader usage of SCADE, different modeling styles came up and also the desire to benefit from the Design Verifier.

2.3 Formal Verification Using SCADE DV

The behavior of a SCADE design model can be given as a transition system on which SAT-based model checking can be performed in order to verify reachability properties[4], see [1].

In contrast to other approaches, the SCADE DV does not offer a temporal logic but properties have to be modeled as *synchronous observers* using the same language operators as for the design. Nevertheless, we will use CTL as a succinct notation for reachability properties to be verified, but the reader should keep in mind that they are encoded as observer nodes at the SCADE level (cf. Figures 1 and 5). Notice, however, that more general temporal properties, notably unbounded liveness, cannot be automatically verified using this SAT-based approach.

2.4 Deductive Cause Consequence Analysis

An important step in safety analysis concerns determining the causal relationship between component fault modes and hazards. There are various techniques that exploit formal methods, notably model checking, for identifying the desired cause-consequence relation [1,16,5]. Here we consider DCCA by Ortmeier et al. [19,11], which is a formal approach to safety analysis that generalizes FTA and FMEA. In DCCA, a hazard H is specified as a state predicate. Primary component fault modes are modeled by adding simple fault automata that are triggered by a Boolean input indicating the occurrence of this particular fault. The immediate effect of a fault on a component is specified in a so-called fault node within the component model. The hazard is implemented as an

[3] SCADE uses the SAT solver developed by Prover Technologies, see `www.prover.com`

[4] In the area of model checking, reachability properties are often called *safety properties*, because they express that the system always stays in a good or "safe" state. In contrast, for us a safety property is any constraint to ensure dependable behaviour.

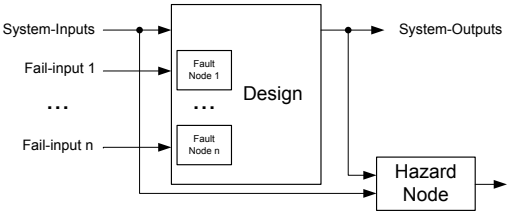


Fig. 1. Integration of DCCA with SCADE

observer node that takes signals from the system and evaluates them according to the *negation* of the hazard predicate H, i.e., returning false whenever the hazard occurs. Figure 1 gives a schematic picture of the integration of DCCA with SCADE.

A core notion of FTA is that of a *critical cut set* of faults for a hazard, i.e. a subset of primary faults for which some operational condition and occurrence sequence exist such that the hazard occurs without further influence. This was formalized in DCCA to the notion of a *critical set*. According to [11], critical sets can be formalized for the SCADE semantics as follows:

Definition 2.1. *Let Γ be a subset of the set of primary component faults Δ of a system* Sys. $\mathsf{Sys}\rangle_{\overline{\Gamma}}$ *denotes that behavior of* Sys *where no fault from $\Delta \setminus \Gamma$ ever occurs. Now Γ is called* critical *for a hazard described by the state predicate H iff* $\mathsf{Sys}\rangle_{\overline{\Gamma}} \not\models \mathbf{AG}\,\neg H$. *A critical set Γ is called* minimal *iff no proper subset of Γ is critical.*

If the analysis reveals a singleton as minimal critical set, this indicates that there exists a single-point-of-failure in the system, which has to be eliminated by further safety measures. A DCCA is called *complete* iff all minimal critical sets have been identified.

Theorem 2.2 (Minimal-Critical-Set Theorem [19,11]). *If a complete DCCA has been conducted for a hazard H then for each minimal critical set, preventing* one *fault from occurrence will prevent the hazard H.*

In order to determine the minimal critical sets we use an iterative approach similar to [1]: Suppose that Δ is the set of all component fault modes identified during the FMEA and that H is a hazard. We iterate over all subsets $\Gamma \subseteq \Delta$ starting with $\Gamma = \emptyset$ and increasing Γ stepwise (singletons, doubletons etc.). We use the SCADE DV to prove whether Γ is critical for H: all trigger inputs for fault modes in Γ become system inputs, and the inputs of all other fault mode nodes are constantly false. Whenever SCADE DV returns a counterexample, i. e. a situation in which the hazard node expressing $\neg H$ is false, we have identified a minimal critical set Γ, and due to the monotonicity of criticality no super set of Γ needs to be considered.

3 Liveness and Abstraction Techniques

In this section we describe the general steps we took in our case study. Here we assume that we are given a design model of the system Sys in a formalism providing support for formal verification. Our steps were as follows:

(1) Given the system model we performed a safety analysis identifying hazards and component fault modes, cf. Section 2.1.

(2) We used DCCA to determine the cause-consequence relationship between component fault modes and hazards. For this we use the SCADE DV to obtain minimal critical sets of fault modes, cf. Section 2.4.

(3) We performed an analysis of a liveness property arising from the safety requirements of the system.

(4) We applied several strategies in order to deal with complexity issues arising in connection with the model checking performed in steps (2) and (3).

We already explained methodological details of steps (1) and (2) in Section 2. In the remainder of this section we will describe steps (3) and (4) in more detail.

3.1 Liveness Analysis

This step of our case study concerns the analysis of a safety related liveness property derived from a system requirement, see Section 4.3 for the concrete formulation. Roughly, this requirement states that a certain non-safe state of the system may occur, but this state must not be permanent, and it must be left within a certain time interval that depends on the system's configuration. Indeed, our analysis in step (2) revealed that this non-safe state of the system will occur under a certain fault mode. This is still in accordance with the requirement. However, it has to be verified that the system will leave this non-safe state in time.

If φ is a propositional formula describing the non-safe state then this property might be formulated in a temporal logic like CTL as $\mathsf{AG}(\varphi \rightarrow \mathsf{AF}\neg\varphi)$. However, SCADE does not provide a direct way to express a liveness property with an unbounded quantifier like AF. A concrete time limit – and hence a bound for the number of cycles – is not specified, but it is individually set for each configuration.

Thus, we explore the model and try to establish a lower bound on the number n of cycles after which the non-safe state is left: It is clear that once in the non-safe state the system will remain there for at least one cycle, and so we verify whether the non-safe state is left after $n = 2, 3, \ldots$. This can be modeled as a SCADE observer by using the operator ImpliesWithinNTicks from the design verifier library, see Figure 2 for $n = 4$.

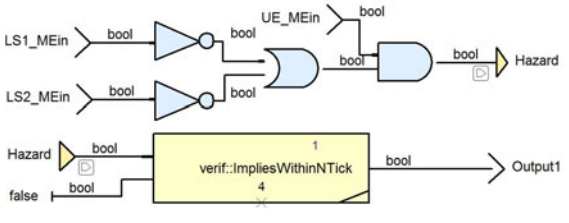


Fig. 2. Observer node for the liveness analysis with $n = 4$

3.2 Dealing with Complexity within SCADE DV

Unfortunately, we failed to complete our analysis just using SCADE DV even with small numbers n due to complexity problems, see Section 4 for details. We will now mention the strategies we have applied in order to deal with this problem.

Abstraction. First we generated an abstract version of the design model in SCADE. For this we applied three different techniques: data abstraction, cone of influence reduction and symmetry reduction (see e. g. [7] for an introduction) – the way in which these are applied is detailed in Section 4. However, even for the abstracted model the liveness verification could not be completed using SCADE DV.

Compositional verification. In a further attempt to cope with complexity of the liveness analysis we manually broke down the given liveness property into proof obligations for each component of the model. We then argued that the liveness property holds for the whole model if the components satisfy their respective proof obligations. This divide-and-conquer strategy allowed us to complete the verification for all but one model component with the SCADE DV.

3.3 Model Transformation to UPPAAL

In order to complete the analysis for this remaining system component we decided to transform this component to another formalism. Manual inspection of the component revealed that its behavior is to a large extent governed by 14 timers that are external to the SCADE model and are provided from the runtime environment in form of an integer input. For this reason we consider it promising to transform that component to a modeling formalism that also takes time into account. We have therefore chosen timed automata [3] with UPPAAL as modeling and analysis tool.

We will briefly describe the principles of the model transformation: We assume that the given SCADE node comes as a *flat* safe state machine. That means that there are no hierarchical states and no memory within states (such as fby operators). Outputs are assigned within the states. We also assume that all transitions in the safe state machine are *strong*, i. e., in a cycle where the transition guard holds the target state of the transition is active (see [4]). These restrictions hold for the model in our case study. For other models, preprocessing steps like flattening of hierarchical states have to be applied.

In contrast to SCADE's deterministic behavior, UPPAAL models may also behave non-deterministically. Thus, the transformation should ensure the deterministic behavior of the model in UPPAAL. The following points are important:

Activation conditions and output variables. The activation condition of a transition in SCADE gets mapped to the guard of the corresponding edge in UPPAAL. If an output flow of a SCADE node changes when taking a transition, we add the corresponding assignment along the translated transition in UPPAAL.

Firing transitions. When a transition guard holds, the corresponding transition in the translated UPPAAL model must fire immediately to faithfully reflect the transition behavior in the SCADE model. For this purpose we use UPPAAL urgent channels. Each translated transition synchronizes on an urgent channel that can always be activated so that no delay occurs along the transition, see Figure 8, where this urgent channel is go2.

Transition priorities. We must make sure no two transition guards are true simultaneously. SCADE uses explicit transition priorities to prevent this. Suppose that we have two transitions with the same source state and with guards φ and ψ, respectively, such that the φ transition has higher priority. Then in the transformed UPPAAL model the first transition has guard φ and the second one the guard $\psi \wedge \neg\varphi$.

Timers. As already mentioned, our SCADE model is partly governed by external timers that are started by certain model outputs and that trigger transitions. In the translated UPPAAL model these timers are explicitly modeled as shown in Figure 3. The timeouts are taken from the SCADE model. The edge between $S0$ and $S1$ will reset the clock variable x. During the urgent state $S1$ clock x does not progress. So any output assignment that may happen together with the start of the timer will be performed as a variable assignment at the edge from $S0$ to $S1$. The invariant $x \leq 10$ makes sure that the state $S1$ is left before x reaches 10. Upon timeout ($x == 10$) we progress to $S3$ and perform any output assignment associated with the timeout along the edge from $S2$ to $S3$.

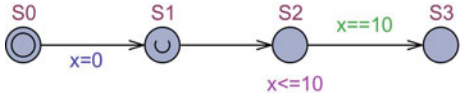


Fig. 3. Modelling a timeout in UPPAAL

This part of the model abstraction realizes the time abstraction mentioned in the introduction. A situation where the SCADE model is waiting for a timeout, i.e., the integer input corresponding to system time increases for a (possibly large) number of cycles where no reaction of the model happens and no model output changes, corresponds to only one transition in the transformed UPPAAL model. This fact leads to a significant reduction in the size of the state space of the transformed model, and we believe this makes formal verification feasible.

3.4 Correctness of the Model Transformation

Although the transformation from SCADE to UPPAAL can be automated in principle, we manually performed it in our case study. We also did not provide a formal proof of the semantic correctness of our transformation. Both tasks were out of the scope of our current project, and so we leave them for future work.

However, in order to establish confidence in the correctness of our transformation we followed an approach based on testing that we will now describe. From the requirements specification we created a test suite. By using the SCADE model test coverage facility this test suite was shown to yield 90% decision coverage of the original SCADE model. The same test suite was used on the translated UPPAAL model. For each test case we recorded the traces (i.e., the list of input and output values in each step) of the simulation of both models.

Now in order to establish the equivalence of the two models we need an appropriate notion of equivalence for the traces. Due to the different timing concepts of the two underlying formalisms, the traces of the SCADE model do not correspond one-to-one to the traces of the translated UPPAAL model. Instead we use a version of *stuttering equivalence*, see e. g. [17,7]. We write $(\vec{v}_1, \ldots \vec{v}_n)$ for a trace of length n produced by a model simulation running some test case. We say that this trace is *stuttering equivalent*

to some other trace $(\vec{w}_1, \ldots, \vec{w}_m)$ if there are sequences $1 = i_1 < i_2 \cdots < i_{k-1} < i_k = n$ and $1 = j_1 < j_2 \cdots < j_{k-1} < j_k = m$ with $k \leq n, m$ and such that

$$\vec{v}_{i_r} = \vec{v}_{i_r+1} = \cdots = \vec{v}_{i_{r+1}-1} = \vec{w}_{j_r} = \vec{w}_{j_r+1} = \cdots = \vec{w}_{j_{r+1}-1} \quad \text{and} \quad \vec{v}_n = \vec{w}_m,$$

where $1 \leq r \leq k-1$. In other words, two lists containing the same input/output values in the same order (but possibly repeating certain list elements a different number of times) are equivalent. Example: Suppose we have a model with one integer input and one Boolean output. Then the two lists $((1, \mathsf{true}), (1, \mathsf{true}), (42, \mathsf{false}), (42, \mathsf{true}))$ and $((1, \mathsf{true}), (42, \mathsf{false}), (42, \mathsf{false}), (42, \mathsf{true}))$ are equivalent.

With this notion of equivalence we compared traces of SCADE simulations of a test case with those of UPPAAL simulations of the same test case. We showed equivalence of the two models with respect to all test cases from our test suite.

4 Comparative Safety Analysis of Level Crossing Control

Now we present the results of our industrial case study. For the sake of brevity we omit some details; they can be found in [8]. Our formal safety analysis was performed comparatively for two SCADE models for the modular level crossing controller in [13]. The two SCADE models were developed by different developers with different implementation styles: the first model uses a design based on safe state machines (SSM) [4] and the second one uses data flow diagrams, which are essentially graphical respresentations of LUSTRE [12]. Whenever information has to be stored for the next execution cycle in the data-flow oriented model, this is done within local variables for which the value is kept using the fby-operator. Both models implement the same architecture with the following operators, which can be composed in order to obtain a level crossing control logic for a specific level crossing layout:

- *route controller* (LC_Route)
- *site controller* (LC_Site)
- *group controller* (LC_LS_Group)
- *time controller* (LC_Timer)

The route controller monitors the activation of the lights and barrier groups depending on the activation signal from a particular route, cf. Figure 4.

The site controller synchronizes all route controllers and group controllers and acts as a logical connector. The group controller controls the lights and barriers, which are grouped logically, using a hardware abstraction layer. Finally, the time controller monitors the time elapsed since the last activation of the LC until its complete deactivation. Depending on the level crossing layout, numbers of inputs, outputs and nodes in the models can vary. In our case, models are specified for controlling of a level crossing with 2 routes and 2 lights-barrier groups, see Figure 4. They have 18 boolean, 1 integer input variables, 5 boolean output variables and 14 constants. The models are composed of 5 nodes: 2 LC_Route, 2 LC_LS_Group, 1 LC_Site. Specifically, the first model has 12 states + 23 transitions for each LC_Route operator, 14 states + 23 transitions for each LC_Site operator, 27 states + 51 transitions for each LC_LS_Group operator. In both models, the LC_Timer operator has 2 states.

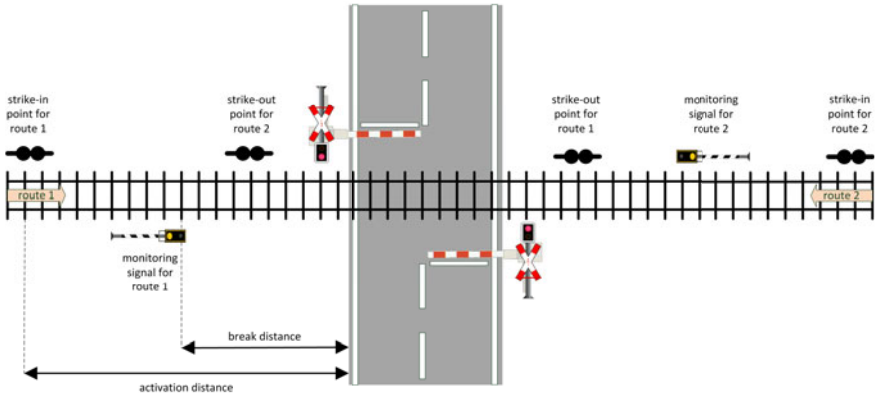


Fig. 4. Sample layout of a locally monitored level crossing

4.1 Application of FMEA

We briefly summarize the results of our FMEA, the details are in [8]. The analysis has revealed ten component fault modes (FM1 – FM10) and two hazards: (a) a train drives through a non-secured level crossing (LC) and (b) car drivers drive against closing/closed barriers. Hazard (a) can happen if the monitoring signal shows the LC to be safe while it is not (i. e., one of the light-barrier groups (LBG) is switched off). That means, if a monitoring signal is activated (*UE_MEin=true*), both LBGs must be switched-on (*LS1_MEin=true* and *LS2_MEin=true*). This state can be represented as a formula H in propositional logic as follows:

$$H = (UE\_MEin \wedge \neg(LS1\_MEin \wedge LS2\_MEin)) \tag{1}$$

Figure 5 shows the SCADE observer node for the hazard H from (1).

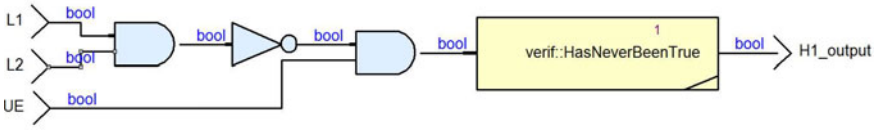


Fig. 5. SCADE model of the hazardous event H

4.2 Application of DCCA

To apply DCCA to our two models we first created an environment model in SCADE simulating stimuli from the hardware of a real level crossing system. We then extended the hardware model with the fault modes and their occurrence patterns from the FMEA.

Functional Correctness. Recall from Section 2.4 that the first step of DCCA considers the empty set of fault modes. So we verify functional correctness of the models

Table 1. Singleton fault modes

	State-based Model			Data-flow Model		
N=1	Valid	Critical	Time	Valid	Critical	Time
FM1		x	1		x	1
FM2	x		122	x		543
FM3	x		14	x		459
FM4	x		13	x		312
FM5	x		143	x		323
FM6	x		108	x		330
FM7	x		199	x		329
FM8	x		14	x		324
FM9	x		15	x		245
FM10	x		14	x		495
Average Time (sec)			64,3			336,1

w.r.t. the safety requirement expressed by the hazard H. With SCADE DV, the proof took 15 seconds for the first model and 322 seconds for the second one. The difference originates from the different implementation styles. The SSM based design seems to have a smaller state space, hence a quicker proof is possible with the first model.

Single Fault Modes. We checked criticality of singleton fault mode sets Γ w.r.t. the hazard node H. Table 1 shows the results and proof execution times in seconds. Only FM1 (unwanted switch off of warning lights) is identified as critical for both models. The rest of the failure modes could be proven to be non-critical.

At Most 2 Fault Modes. This step requires analysis of the fault mode sets Γ with $|\Gamma| = 2$. As FM1 is already critical, we only analyze sets Γ with FM1 $\notin \Gamma$. We assume that for each hardware type only one fault mode can occur at a time, and only a single hardware component of the same hardware type can fail at a time, e. g. only one of the barriers in a barrier group can fail. Finally, sensor failures can occur either as a false detection or a mis-detection, but not both.

Figure 6 and 7 present the analysis results for the first and second model, respectively. The cells marked with an "x" represent single fault modes and already have been treated in the previous step. The *Gray* colored combinations have not been analyzed due to the assumptions explained in the previous paragraph and since FM1 is critical. *Red* colored cells (marked by "C") represent the critical sets while *green* cells (marked by "V⟨time⟩") represent the non-critical fault modes and also indicate the proof durations in seconds. Critical sets were proven to be critical within 1 second. *White* colored combinations (marked by "U") could not be proven to be critical or non-critical in a reasonable time. We could not analyze all relevant 2-element fault mode combinations, hence, our DCCA for critical doubletons is not complete. Since three simultaneous failures are highly unlikely we terminated the analysis with this step.

To summarize, while our analysis remains incomplete we have identified three minimal critical sets: {FM1}, {FM4,FM6} and {FM4,FM7}; FM4 means that a barrier is stuck and FM6 and FM7 mean the corresponding sensor does not detect this.

	FM1	FM2	FM 3	FM4	FM5	FM6	FM7	FM8	FM9	FM10
FM1	x									
FM2		x								
FM3			x							
FM4		V149	V15	x						
FM5		V457	U		x					
FM6		U	U	C	V256	x				
FM7		U	U	C	V14		x			
FM8		U	U	U	U	U	U	x		
FM9		U	U	U	U	U	U		x	
FM10		U	U	U	U	U	U			x

Fig. 6. Results for the state-based model

	FM1	FM2	FM 3	FM4	FM5	FM6	FM7	FM8	FM9	FM10
FM1	x									
FM2		x								
FM3			x							
FM4		U	U	x						
FM5		U	U		x					
FM6		U	U	C	U	x				
FM7		U	U	C	U		x			
FM8		U	U	U	U	U	U	x		
FM9		U	U	U	U	U	U		x	
FM10		U	U	U	U	U	U			x

Fig. 7. Results for the data-flow model

4.3 Liveness Analysis

We have shown that both models do have critical sets of fault modes. From a control engineering point of view this was expected to happen, and rather than requiring prevention of the hazard the corresponding safety requirement for the system reads as follows:

Requirement. *A state in which the monitoring equipment feedbacks its status as activated* (UE_MEin=true) *and at least one LBG is non-active* (LS1_MEin=false *or* LS2_MEin=false) *is non-safe. Such a state must not be permanent and has to be left independently from the input values as quickly as possible by deactivation of the monitoring signal.*

This requirement can be formalized as a liveness property in CTL as follows:

$$\mathsf{AG}(UE\_MEin \wedge \neg(LS1\_MEin \wedge LS2\_MEin) \Rightarrow \mathsf{AF}(\neg UE\_MEin)) \quad (2)$$

As explained in Section 3.3 we tried to use the SCADE DV with an observer as shown in Figure 2 to obtain a lower bound on the number n of cycles the system remains in the non-safe state corresponding to our hazard property H.

4.3.1 Liveness of the State-Based Model. For $n = 2$, SCADE DV delivered "falsifiable" as an answer within a few seconds, which means that the hazard may last for at least two cycles. For $n = 3$ it was not possible to prove the correctness of the system after 1 week of execution time as a result of the state explosion problem. To overcome this problem we used several strategies (see Section 3.3). Full details are in [8].

Data Abstraction. The only integer input of the models is used for synchronization of the controller with a global clock *T_System*. The value of *T_System* is used by 14 different timers for realizing the delays and timeouts. For the current verification, only 10 of them are relevant. Using SCADE assumptions we have put bounds on the possible values of *T_System*, thus reducing the complexity of the verification.

Cone of Influence Reduction. Only 7 inputs out of 19 of the model have an influence on the observer node. Some of the remaining inputs have constant values as they correspond to configuration settings. These inputs can be replaced by the constants. As a consequence some transition triggers are simplified or even become constantly false. As a result some states become unreachable and are removed as well.

Symmetry Reduction. Here we remove the identical operators and symmetric configurations in order to reduce the state space. Our initial configuration was a 2 Route+2 LBG level crossing. Obviously, a reduced model (1 Route+1 LBG) needs to satisfy the liveness property, too. If we can verify the validity, it can substantiate the claim that the full system is correct.

Unfortunately, even with the above three simplifications it was still impossible to verify the liveness property with the SCADE DV.

4.3.2 Compositional Verification. With an operator level analysis, we identified that an occurrence of FM1 generates an *LS_MEin=false* (an LBG is not switched on) signal, which is converted to *AN_MEin=false* (not all LBG's are switched on) signal through an *AND* gate by the *LC_LS_Group* operator. It feedbacks this status to the *LC_Site* operator, which updates an internal variable as *LS_MEin=false*. Finally, *LC_Route* receives this signal and signals the monitoring signal to switch-off. This is followed by a *UE_MEin=false* signal by a failure-free monitoring signal. This led us to decompose the liveness property for the whole model into properties for its components:

1. For *LC_LS_Group* operator: *LS_MEin=false* shall be followed by an *AN_MEin=false* signal; in CTL:

$$\mathsf{AG}((\neg LS\_MEin \wedge AN\_MEin) \Rightarrow \mathsf{AF}(\neg AN\_MEin)). \tag{3}$$

2. For the *LC_Site* and *LC_Route* operators: *AN_MEin=false* shall be followed by an *UE_MEin=false* signal; in CTL:

$$\mathsf{AG}((UE\_MEin \wedge \neg AN\_MEin) \Rightarrow \mathsf{AF}(\neg UE\_MEin)).$$

It can be easily seen that both parts together imply the main liveness property. Analysis of the second formula was successful. Thus, the operators *LC_Site* and *LC_Route* are verified. They were found to satisfy the liveness property within 6 execution cycles. However, the first part could not be proven in a reasonable time. This led us to consider the *LC_LS_Group* operator as the bottleneck for the complete analysis.

4.3.3 UPPAAL Transformation. Using the model transformation described in Section 3 we transformed the *LC_LS_Group* operator to UPPAAL. The resulting UPPAAL model is a network of 5 timed automata. Four automata model the environment such as hardware behavior, switch-on and switch-off commands and the fault mode FM1. The *LBG automaton* (see Figure 8) models the behavior of the LC_LS_Group operator. It synchronizes with other UPPAAL automata over the urgent *go1* channel. This channel ensures progress and an immediate transition as soon as an edge's guard holds. For example, it can be seen that edge from *S00* to *S01* fires as soon as *AN_SEin* holds.

Verification with UPPAAL. UPPAAL uses a fragment of timed CTL (TCTL) [2]. Like TCTL, the query language consists of path formulæ and state formulæ [20]. In our case the liveness property (3) for the *LC_LS_Group* operator will be rewritten as:

```
(not LS_MEin and AN_MEin) --> not AN_MEin
```

This property means when a failure occurs in the hardware (*LS_MEin=false*), the output variable *AN_MEin* of the LBG automaton will eventually be false. This property is valid and the proof has been completed within seconds.

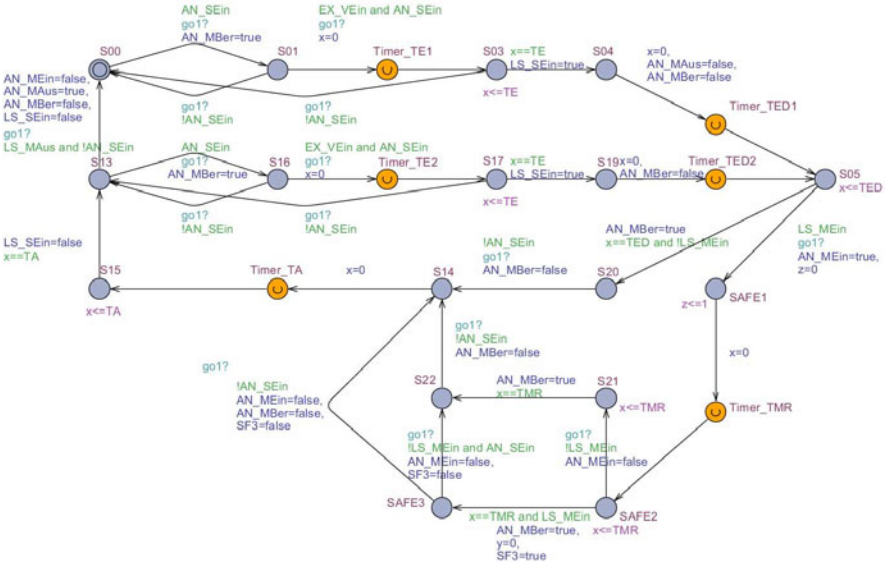


Fig. 8. LBG automaton in UPPAAL

4.3.4 Liveness of the Data-Flow Design. The second model could not be proven correct w.r.t. the liveness property with the full configuration (2 Route+2 LBG). Its data-flow oriented design does not allow us to apply state and transition based abstraction techniques. However, symmetry reduction is still possible. A model with reduced configuration (1 Route+1 LBG) was found to be valid for 4 cycles within 130 seconds.

In contrast to the first model, in the case of the second model the liveness property could be proven to be valid for a reduced model without decomposing it. As a possible reason we consider the direct connection between LBG hardware elements and the *LC_Route* operator. In this model, whenever a failure occurs, it is directly sensed by the route controller that controls the monitoring signal. This connection eliminates the effects of the other operators on the liveness property. Hence, this property is not dependent on the feedbacks from the LC_LS_Group operator.

5 Lessons Learned and Conclusions

We performed a comparative case study on formal safety analysis and verification. Our starting point were two functional equivalent SCADE models of a level crossing controller software, both developed in industry. As a formally founded, model-based approach was used in the design already, a seamless expansion towards formal safety analysis and verification within the same tool environment was straightforward. Despite of the difficulties we faced and which fill major parts of our experience report, we were able to identify some of the most relevant critical sets of fault modes as well as to verify numerous safety requirements. In addition, confidence that the results of the formal verification apply for the finally generated executable is strong, since the target

code is automatically generated from the SCADE models by the certified code generator. Such a full integration and tool support for design, code generation and verification is mandatory for the successful usage of formal methods in industrial development of safety-critical systems. In this way formal methods can contribute with strong evidence to a safety case a system manufacturer has to provide to certification authorities.

5.1 Lessons Learned

Our verification results support the hypothesis that applicability of formal analysis based on SAT model checkers depends on the design and modeling style. We found that the state-based model lends itself better to abstraction and reduction techniques than the data-flow oriented one; in our case study some properties were only provable after reduction and abstraction had been applied. Another argument pro state-based designs is that they facilitate transformation to other model checkers. For data-flow oriented designs some abstraction techniques do not seem obviously applicable.

On the other hand, we found that our liveness analysis could be performed with the data-flow oriented model after a symmetry reduction with the SCADE DV whereas for the state-based model further abstraction and a model transformation to UPPAAL was necessary. In this concrete case this seems to be due to architectural differences. However, from our case study we cannot conclude that data-flow oriented models are in general better suited for formal verification. We faced severe complexity issues during DCCA and liveness analysis for both of our moderately sized models.

After all, we cannot fully explain why the two modeling styles differ w.r.t. verification: The size and complexity of the models are similar. But in the data-flow design the state information is scattered throughout the model and less uniform than in the state-based one. That may be a reason why the model transformation to SCADE DV's SAT-based model checker yields a better result for the state-based model. But to substantiate this assumption the SCADE internal model transformation would have to be inspected in detail.

5.2 Open Issues and Future Work

Next, we discuss a few open issues: Firstly, the heuristics we applied in order to deal with complexity issues during our liveness analysis also could be applied to the formal verifications that happen during the DCCA. Secondly, our model transformation to UPPAAL makes heavy use of the specific form of the given SCADE model (flat state machine, timers). If one aims at an extension to safe state machines *with* hierarchy it is questionable to which extent the flattening that has to be part of any (automatic) transformation will blow-up the resulting timed automata and whether formal verification in UPPAAL will perform well on them. Perhaps, it is possible to exploit compositional techniques in this case, and we leave this as an open question. Thirdly, we believe that the steps we performed in our case study are successfully applicable more generally provided that given SCADE models adhere to similar restrictions. Lastly, we verified our model transformation with the help of testing and did not provide of formal proof of the semantic correctness of our model transformation. Such a correctness proof would, of course, be desirable, but here we leave this for future work.

Finally, we have to state that formal verification is still constrained by scalability problems that hampers its usage in practice. We solved them for the state-based model by transforming it into UPPAAL timed automata, a formalism that offers a much more efficient handling of time. From a methodological point of view, this model transformation is a *time* abstraction. However, it cannot easily be integrated with the synchronous modeling paradigm SCADE is based on. In addition, thorough expertise on verification techniques, the underlying semantics and algorithmics is a pre-requisite to come up with an alternative formalism that potentially can efficiently solve a specific verification problem. However, a desirable improvement of todays verification engines is feedback that, in case of complexity problems, directs the developer to the origin of the problems within the models. For example, a measure of the impact of selectable model elements or system components on state space size or other complexity measures for the verification problem will be of great help, as it will ease the choice for a promising abstraction or reduction heuristic.

Acknowledgements. We are grateful to Siemens AG, I MO RA, for providing the real industrial models for our formal analysis and to Dr. Dirk Peter for fruitful discussions.

References

1. Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing Safe, Reliable Systems Using Scade. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 115–129. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11925040_8
2. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
4. André, C.: Semantics of S.S.M (safe state machine). Tech. Rep. UMR 6070, I3S Laboratory, University of Nice-Sophia Antipolis (2003)
5. Bozzano, M., Villafiorita, A.: Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform (2003)
6. CENELEC: EN 50128 – Railway Applications – Software for Railway Control and Protection Systems. European Standard (2001)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
8. Daskaya, I.: Comparative Safety Analysis and Verification for Level Crossings. Master's thesis, Technische Universität Braunschweig (2011)
9. DIN: EN 50126: Spezifikation und Nachweis der Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit und Sicherheit, RAMS (1999)
10. DIN: EN 50129: Bahnanwendungen – Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Sicherheitsrelevante elektronische Systeme für Signaltechnik (2003)
11. Güdemann, M., Ortmeier, F., Reif, W.: Using deductive cause-consequence analysis (DCCA) with SCADE. In: Saglietti, F., Oster, N. (eds.) SAFECOMP 2007. LNCS, vol. 4680, pp. 465–478. Springer, Heidelberg (2007)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (1991)

13. Hanisch, H.M., Pannier, T., Peter, D., Roch, S., Starke, P.: Modeling and formal verification of a modular level-crossing controller design (2000)
14. IEC 60812: Analysis techniques for system reliability (2006)
15. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements (1998), corrigendum (1999)
16. Joshi, A., Whalen, M.: Modelbased safety analysis: Final report. Tech. rep., NASA (2005)
17. Lamport, L.: What good is temporal logic. Information Processing 83, 657–668 (1983)
18. McDermid, J.A., Nicholson, M., Pumfrey, D.J., Fenelon, P.: Experience with the application of HAZOP to computer-based systems. In: 10th Annual Conference on Computer Assurance (Compass), pp. 37–48 (1995)
19. Ortmeier, F., Reif, W., Schellhorn, G.: Deductive cause consequence analysis (DCCA). In: Proc. IFAC World Congress. Elsevier, Amsterdam (2006)
20. UPPAAL 4.0: Small Tutorial (November 16, 2009), http://www.it.uu.se/research/group/darts/uppaal/smalltutorial.pdf

Structural Test Coverage Criteria for Integration Testing of LUSTRE/SCADE Programs*

Virginia Papailiopoulou[1], Ajitha Rajan[2], and Ioannis Parissis[3]

[1] INRIA Rocquencourt
[2] Université Joseph Fourier - Laboratoire d'Informatique de Grenoble
[3] Grenoble INP - Laboratoire de Conception et d'Intègration des Systèmes
virginia.papailiopoulou@inria.fr, ajitha.rajan@imag.fr,
ioannis.parissis@grenoble-inp.fr

Abstract. LUSTRE is a formal synchronous declarative language widely used for modeling and specifying safety-critical applications in the fields of avionics, transportation, and energy production. In such applications, the testing activity to ensure correctness of the system plays a crucial role in the development process. To enable adequacy measurement of test cases over applications specified in LUSTRE (or SCADE), a hierarchy of structural coverage criteria for LUSTRE programs has been recently defined. A drawback with the current definition of the criteria is that they can only be applied for unit testing, i.e., to single modules without calls to other modules. The criteria experiences scalability issues when used over large systems with several modules and calls between modules. We propose an extension to the criteria definition to address this scalability issue. We formally define the extension by introducing an operator to abstract calls to other modules. This extension allows coverage metrics to be applied to industrial-sized software without an exponential blowup in the number of activation conditions. We conduct a preliminary evaluation of the extended criteria using an Alarm Management System.

1 Introduction

LUSTRE [4,1] is a synchronous dataflow language widely used to model reactive, safety-critical applications. It is based on the synchronous approach where the software reacts to its inputs instantaneously. LUSTRE also forms the kernel language for the commercial toolset, SCADE (Safety Critical Application Development Environment[1]), used in the development of safety-critical embedded software. SCADE has been used in several important European avionic projects such as AIRBUS A340-600, A380, EUROCOPTER.

The verification and validation activity ensuring the correctness of the system is very important for applications in the safety-critical systems domain. For applications modeled in the LUSTRE language, this concern has been addressed

* This research work is supported by the SIESTA project (www.siesta-project.com), funded by ANR, the French National Research Foundation.

[1] www.esterel-technologies.com

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 85–101, 2011.

either by means of formal verification methods [5] or using automated testing approaches [7,9]. State space explosion still poses a challenge for applying formal verification techniques to industrial applications. Hence, developers use intense testing approaches to gain confidence in the system correctness.

To determine whether the testing process can terminate, several test adequacy criteria based on the program control flow have been used in the past, such as branch coverage, LCSAJ (Linear Code Sequence And Jump) [10] and MCDC (Modified Condition Decision Coverage) [2]. These criteria do not conform to the synchronous data-flow paradigm and when applied do not provide meaningful information on the Lustre specification. To tackle this problem, Lakehal et al.[6] and Papailiopoulou et al.[8] defined structural coverage criteria specifically for LUSTRE specifications. The structural coverage criteria for LUSTRE are based on *activation conditions* of paths. When the activation condition of a path is *true*, any change in the path entry value causes modification of the path exit value within a finite number of time steps.

The LUSTRE coverage criteria defined in [6] and [8] can only be used for unit testing purposes, i.e. adequacy can only be measured over a single LUSTRE node (program units). To measure adequacy over Lustre nodes with calls to other nodes, the calls would have to be inline expanded for coverage measurement. Doing this results in an exponential increase in the number of paths and activation conditions to be covered. As a result, the current definition of the criteria does not scale to large and complex systems.

To tackle this scalability issue, we extend the definition of Lustre coverage criteria to support node integration. In our definition, we use an abstraction technique that replaces calls to nodes with a new operator, called the *NODE* operator. This abstraction avoids the entire set of paths of the called node from being taken into account, and instead replaces it with a unit path through the *NODE* operator. We evaluated the extended criteria over an Alarm Management System used in the field of avionics.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the essential concepts of the LUSTRE language and the existing coverage criteria defined for LUSTRE programs. In Section 3, we define the extended coverage criteria for integration testing, and Section 4 presents a preliminary evaluation.

2 Background

2.1 Overview of LUSTRE

A LUSTRE program is structured into nodes. Nodes are self-contained modules with inputs, outputs, and internally-declared variables. A node contains an unordered set of equations that define the transformation of node inputs into outputs through a set of operators. Once a node is defined, it can be called (or instantiated) within other nodes like any other expression in the specification.

In addition to common arithmetic and logical operators (`+`, `-`, `*`, `/`, `and`, `or`, `not`), LUSTRE supports two temporal operators: *precedence* (`pre`) and *initialization*

```
node Never(A: bool) returns (never_A: bool);
let
  never_A = not(A) -> not(A) and pre(never_A);
tel;
```

	c_1	c_2	c_3	c_4	...
A	false	false	true	false	...
never_A	true	true	false	false	...

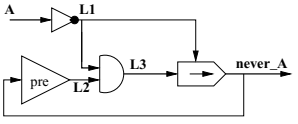


Fig. 1. A LUSTRE node and its operator network

(->). The `pre` operator introduces a delay of one time unit, while the -> operator, also called *followed by* (`fby`), allows the initialization of a sequence. Let $X = (x_0, x_1, x_2, x_3, \ldots)$ and $E = (e_0, e_1, e_2, e_3, \ldots)$ be two LUSTRE expressions. Then `pre`(X) denotes the sequence $(nil, x_0, x_1, x_2, x_3, \ldots)$, where *nil* is an undefined value, while X->E denotes the sequence $(x_0, e_1, e_2, e_3, \ldots)$.

Figure 1 illustrates a simple LUSTRE program and an instance of its execution. This program has a single input boolean variable (`A`) and a single output boolean variable (`never_A`). The output is *true* if and only if the input has never been *true*. Lustre nodes are usually represented by an *operator network*, a labeled graph connecting operators with directed edges. An operator in the network specifies data-flow transfers from inputs to outputs. An edge specifies the data-flow between two operators. There are three kinds of edges: input, output and internal edges. Input (or output) edges are occurrences of input (or output) variables of the LUSTRE node. Internal edges correspond to occurrences of local variables. The operators *not, and, pre,* and *followed by* in the operator network of Figure 1 are connected so that they correspond exactly to the sequence of operations in the Lustre node *Never* in Figure 1. At the first execution cycle, the output `never_A` from the *followed by* operator is the input edge L1 (negation of input `A`). For the remaining execution cycles, the output is the input edge L3 (conjunction of previous `never_A` and the negation of `A`).

2.2 LUSTRE Coverage Criteria

Given an operator network N, paths in the program can be viewed as possible directions of flows from the input to the output. Formally, a path is a finite sequence of edges $\langle e_0, e_1, \ldots, e_n \rangle$, such that for $\forall i \epsilon\, [0, n-1]$, e_{i+1} is a successor of e_i in N. A *unit path* is a path with two successive edges $\langle e_i, e_{i+1} \rangle$. For instance, in the operator network of Figure 1, we can find the following paths.

$p_1 = \langle A, L_1, never\_A \rangle$; $p_2 = \langle A, L_1, L_3, never\_A \rangle$;
$p_3 = \langle A, L_1, never\_A, L_2, L_3, never\_A \rangle$;
$p_4 = \langle A, L_1, L_3, never\_A, L_2, L_3, never\_A \rangle$

All these paths are *complete paths*, because they connect a program input with a program output; the paths p_1 and p_2 are *elementary paths*, because they contain no cycles while the paths p_3 and p_4 contain one cycle[2]. Lakehal et al.[6]

[2] Cyclic paths contain one or more pre operators.

Table 1. Activation conditions for Boolean/Relational/Conditional operators

Operator	Activation condition
$s = NOT\,(e)$	$AC\,(e, s) = true$
$s = AND\,(a, b)$	$AC\,(a, s) = not\,(a)\ or\ b$; $AC\,(b, s) = not\,(b)\ or\ a$
$s = OR\,(a, b)$	$AC\,(a, s) = a\ or\ not\,(b)$; $AC\,(b, s) = b\ or\ not\,(a)$
$s = ITE\,(c, a, b)$	$AC\,(c, s) = true$; $AC\,(a, s) = c$; $AC\,(b, s) = not\,(c)$
$s = op\,(e)$, where $op \epsilon \{<, >, \leq, \geq, =\}$	$AC\,(e, s) = true$
$s = op\,(a, b)$, where $op \epsilon \{+, -, \star, /\}$	$AC\,(a, s) = true$ $AC\,(b, s) = true$

Table 2. Activation condition definitions using path prefix p' and last operator

Last Operator	Activation condition
$Boolean/Relational/Conditional$	$AC\,(p) = AC(p')$ and $OC(e_{n-1}, e_n)$
$FBY\,(init, nonInit)$	$AC\,(p) = AC(p')$ -> $false$ for initial cycle $AC\,(p) = false$ -> $AC(p')$ for all but the initial cycle
$PRE\,(e)$	$AC\,(p) = false$ -> $pre\,(AC(p'))$

only considered paths of finite length. A path is considered finite if it contains no cycles or if the number of cycles is limited.

Lakehal et al. associate a notion of an *activation condition* with each path. When the activation condition of a path is true, any change in the path entry value causes eventually the modification of the path exit value. A path is activated if its activation condition has been true at least once during an execution. Table 1 summarizes the formal expressions of the activation conditions for boolean, relational, and conditional LUSTRE operators. In this table, each expression $s = op\,(e)$ in the first column, that uses the operator `op` with the input e and output s, is paired with the respective activation condition $AC\,(e, s)$ for the unit path $\langle e, s\rangle$ formed with operator `op`. Activation condition for a unit path with the `NOT` operator is always *true*, since the output is always affected by changes in the input. For operators with more than one input, such as $AND(a, b)$, there will be more than one unit path –$\langle a, s\rangle$and$\langle b, s\rangle$– from each input to the output. The activation conditions for such operators are listed from each operator input to each output. To exemplify, consider the activation conditions for $AND(a, b)$ –$AC(a, s)$, and $AC(b, s)$. The activation condition for the unit path from input a to output s, $AC(a, s)$, is true when the effect of input a propagates to the output s. This happens when input a is $false$, in which case the output is $false$ regardless of the value of input b. It also occurs when input b is *true*, the output value solely depends on the value of input a. As a result, $AC(a, s) = not(a)\ or\ b$. The activation condition, $AC(b, s)$, from input b is computed in a similar fashion.

For a given path p of length n, $\langle e_1, ..., e_{n-1}, e_n\rangle$, the activation condition $AC(p)$ is recursively defined as a function of the last operator op in it ($e_{n-1} \in$

$in(op)$ and $e_n \in out(op)$) and its path prefix p' of length $n-1$,$\langle e_1, ..., e_{n-1}\rangle$. If p is a single edge ($n = 1$), $AC(p)$ is *true*. Table 2 presents definitions of $AC(p)$ based on the last operator *op* and the path prefix p'. Let $OC(e_{n-1}, e_n)$ be the activation condition associated with operator *op* on the unit path (e_{n-1}, e_n).

To illustrate the activation condition computation, consider the path $p_2 = \langle A, L_1, L_3, never\_A\rangle$ in the operator network for the node `Never` in Figure 2. Path p_2 and its sub paths are illustrated in Figure 2. The activation condition for p_2 will be a boolean expression that gives the condition under which effect of input A progresses to output $never\_A$. To calculate this condition, we progressively apply the rules for the activation conditions of the corresponding operators in Table 1 and Table 2. Equations (1), (2), (3), and (4) shown below along with Figure2 illustrate this computation. We start from the end of the path and progress towards the beginning, moving one step at a time along the unit paths.

$$AC\,(p_2) = false \rightarrow AC\,(p')\ \ where\ p' = \langle A, L_1, L_3\rangle\,. \tag{1}$$

$$AC(p') = not(L_1)\ or\ pre(never\_A)\ and\ AC(p'') \tag{2}$$

$$AC(p'') = true \tag{3}$$

Upon backward substitution of values for AC(p') and AC(p'') from Equations 2 and 3, respectively, into Equation 1 we get

$$AC\,(p_2) = false \rightarrow A\,or\,pre\,(never\_A) \tag{4}$$

The activation condition for path p_2 in Equation 4 states that at the first execution cycle, the path p_2 is not activated. For the remaining cycles, to activate path p_2 either the input A needs to be *true* at the current execution cycle or the output at the previous cycle needs to be *true*.

Criteria Definition. Lakehal et al.[6] defined three coverage criteria for Lustre/SCADE programs and implemented a coverage measurement tool called Lustructu based on the definitions. The coverage definitions are for a given finite path length. To understand our contributions in extending the criteria later in the paper, we present the formal definitions of the criteria from [6].

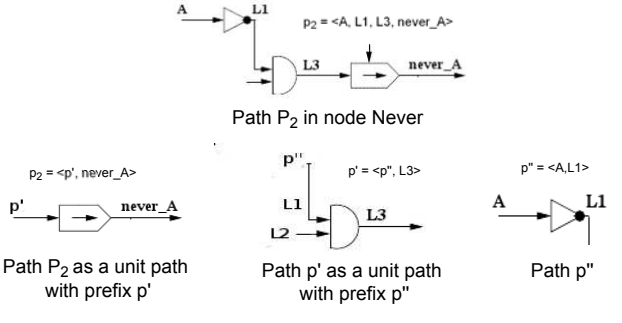


Fig. 2. Path p_2 and its sub paths in node `Never`

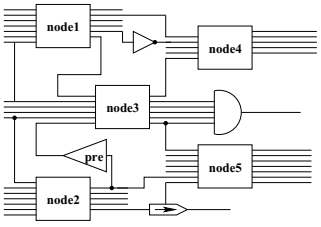


Fig. 3. Example of a complex LUSTRE program

Let $\mathcal{T}$ be the set of test sets (input vectors) and $P_n = \{p|length(p) \leq n\}$ the set of all paths in the operator network whose length is less than or equal to n. The input of a path p is denoted as $in\,(p)$ and a path edge is denoted as e. A family of criteria were defined by Lakehal et al. for a finite path length n:

Basic Coverage Criterion (BC). This criterion is satisfied if there is a set of test input sequences, $\mathcal{T}$, that activates at least once the set P_n. Formally, $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $AC\,(p) = true$. The aim of this criterion is to ensure that all dependencies between inputs and outputs have been exercised at least once.

Elementary Conditions Criterion (ECC). In order to satisfy this criterion for a path p, it is required that the path p be activated for both values, *true* and *false,* of the input (taking only boolean input variables into consideration). Formally, $\forall p \in P_n$, $\exists t_1 \in \mathcal{T}$: $in\,(p) \wedge AC\,(p) = true$ and $\exists t_2 \in \mathcal{T}$: $not\,(in\,(p)) \wedge AC\,(p) = true$. This criterion is stronger than the basic criterion since it also takes into account the impact of input value variations on the output.

Multiple Conditions Criterion (MCC). This criterion checks whether the path output depends on all combinations of the path edges, including the internal ones. To satisfy this criterion, test input sequences need to ensure that the activation condition for each edge value along the path is satisfied. Formally, $\forall p \in P_n, \forall e \in p, \exists t_1 \in \mathcal{T}$: $e \wedge AC\,(p) = true$ and $\exists t_2 \in \mathcal{T}$: $not\,(e) \wedge AC\,(p) = true$.

For a given path length n, [6] shows that MCC_{n} subsumes ECC_{n} which in turn subsumes BC_{n}.

3 Integration Testing Approach

3.1 Motivation

For simple LUSTRE programs, coverage computation can be performed in a short amount of time. If the path length and number of paths is low, the number of activation conditions to be satisfied is also low, and unsatisfied conditions can be easily identified and analyzed by test designers. However, for complex Lustre programs, the number of activation conditions is usually high. This is particularly true for the MCC criterion, where the number of activation conditions to be satisfied increases dramatically with the length and the number of paths. As a result, coverage assessment and analysis become prohibitively expensive making

the criteria inapplicable. One of the primary reasons for the extremely high number of paths and activation conditions in complex Lustre programs is due to the presence of node calls. For instance, Figure 3 partially illustrates a complex node with calls to node1, node2, node3, node4, and node5. In order to measure coverage of the complex node in Figure 3 with the existing definition of Lustre criteria in [6], the node calls would have to be expanded into their definitions. The paths and the corresponding activation conditions are locally computed within each expanded node call, and these are then combined with expressions in the global node to compute the final activation conditions. This often results in a huge number of paths and activation conditions in the global node.

3.2 Path Activation Conditions

To help tackle this issue of intractable number of activation conditions, we define an approximation of the operator network by abstracting the called nodes. We replace calls to nodes with a new operator, `the NODE` operator, in the operator network. The inputs and the outputs of the `NODE` operator are the inputs and the outputs of the called node. This abstraction avoids the entire set of paths of the called node from being taken into account for coverage assessment. However, the abstraction is such that it is still possible to determine whether the output of a called node depends on a given input of the called node. More precisely, as illustrated in Figure 4, at the global level the set of paths beginning from an input e_i and ending at an output s_i, in the operator network of the called node, is represented by a single unit path $p = \langle e_i, s_i \rangle$ using the `NODE` operator. To compute the activation condition of p, we first compute the activation conditions of all paths from the edge e_i to the edge s_i in the called node. We then combine these activation conditions using the disjunction operator into a single activation condition for p.

To exemplify, consider the unit path $p2 = \langle e2, s2 \rangle$ at the global level in Figure 4. To compute the activation condition of $p2$ with the $NODE$ operator $NODE0$, we first compute the activation conditions of paths from the edge $e2$ to the edge $s2$ in the called node. Depending on the number of cycles we consider in the called node, the set of paths from $e2$ (input b) to $s2$ (output d) in the called node is $\{\langle b, d\rangle, \langle b, d, t1, d\rangle, \langle b, d, t1, d, t1, d\rangle, \langle b, t1, d, t1, d, t1, d\rangle ...\}$. We compute activation conditions of each of these paths; $\mathrm{AC}(\langle b, d\rangle)$, $\mathrm{AC}(\langle b, d, t1, d\rangle)$, and so forth. Finally, we compute the activation condition of the unit path $p2$ at the global level as the disjunction of each of these individual activation conditions (depending on the number of cycles considered), $AC(p2) = AC(\langle b, d\rangle) \vee AC(\langle b, d, t1, d\rangle) \vee AC(\langle b, d, t1, d, t1, d\rangle) \vee ...$

We would like readers to note that, even though the proposed abstraction requires the computation of activation conditions for paths in the called node, the computation effort is only increased by a linear amount (by the number of paths from the edge e_i to the edge s_i in the called node). The number of paths and activation conditions in the callee node is *not* affected by the number of paths from the edge e_i to the edge s_i in the called node since the abstraction represents them by a unit path. This, however, is not the case in the original

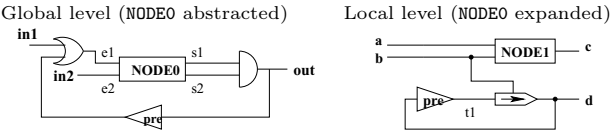


Fig. 4. A LUSTRE node using a compound operator

criteria definition. The original definition substitutes the *entire* operator network of the called node into the callee node, resulting in an exponential increase in the number of paths and activation conditions in the callee node. As a result, our approach still results in significant savings in effort required for coverage assessment. In Section 4, we illustrate the difference in number of activation conditions between the original criteria definitions and our proposed extension using a case study.

A called node may also contain calls to other nodes which in turn may call other nodes and so on. An example of such an occurrence is shown in Figure 4, where at the global level there is a call to NODE0 which in turn calls NODE1. As a result, we get a tree structure of called nodes and every level of this tree corresponds to a different *depth* of node integration. In Figure 4(a), let $AC^0(p)$ be the activation condition of $p = \langle e_1, s_1 \rangle$ at level zero. At level one, since `NODE0` calls `NODE1` (Figure 4(b)), the following expression describes the correlation of activation conditions between the different levels of integration:

$$AC^0(\langle e_1, s_1 \rangle) = AC^1(\langle a, c \rangle) \tag{5}$$

In general, at level (or depth) m, a path activation condition depends on the activation conditions at level $m+1$, as seen in the following definition.

Definition 1. *Let E and S be the sets of inputs and outputs respectively of a* `NODE` *operator at level m such that there is an input $e_i \in E$ and an output $s_i \in S$. Let $n>0$ be any positive integer and $p_1, p_2, \ldots, p_k$ be the paths from the input e_i to the output s_i the length of which is less than or equal to n. The* ***abstract activation condition*** *of the unit path $p = \langle e_i, s_i \rangle$ at depth m and path length n is defined as follows:*

$$AC_n^m(p) = AC_n^{m+1}(p_1) \vee AC_n^{m+1}(p_2) \vee \cdots \vee AC_n^{m+1}(p_k) \tag{6}$$

Similar to the activation condition definition for the basic operators, the abstract activation condition signifies the propagation of the effect of the input value to the output. Disjunction of the activation conditions of the involved paths, ensures that at least one of the dependencies of s_i on e_i is taken into account. In other words, if at least one path of length lower or equal to n in the called node is activated, then the unit path p for the `NODE` operator is activated, $AC_n^m(p) = true$, implying that the value of the output s_i is affected by the value of the input e_i of the `NODE` operator.

3.3 Extended Criteria Definition

We extend the original Lustre coverage criteria for unit testing[6] to support node integration using the abstraction technique discussed in the previous section. We replace calls to nodes with the `NODE` *operator*. We re-define the three classes of path-based criteria—BC, ECC, and MCC—to account for node integration. We use three input parameters in the definition of each criterion,

1. *Depth of integration:* This parameter determines the level at which abstraction of node calls using the NODE operator will begin (i.e where the abstract activation condition will be used). This parameter is not present in the definition of the original criteria.
2. *Maximum path length* from an input to an output: This parameter is used in the original criteria definition. However, the paths to be covered and their length vary depending on the depth of integration parameter.
3. *Number of cycles* to consider in paths: This parameter is used in the original criteria definition to indicate the number of cycles to take into account in paths whose length is at most equal to the maximum path length parameter.

Depth of Integration. Coverage analysis of a node using the parameter $depth = i$ indicates that called nodes up until depth i are inline expanded, while called nodes at depths i and deeper than i are abstracted using the `NODE` operator. The node being covered corresponds to depth 0. For instance, in Figure 5 that represents the call graph of an application with several embedded nodes (used in [6]), covering node #1 using the parameter $depth = 0$ implies that calls to all nodes starting from nodes #2, #3, and #4 (since they are called by node#1) will be abstracted using the `NODE` operator. If we set the depth of integration as $depth = 1$, it implies that calls to nodes #2, #3, and #4 will be inline expanded and all other nodes (that are in turn called by nodes #2, #3, and #4) will be replaced by the `NODE` operator. Developers can choose a value for the integration depth parameter according to the complexity of the system being covered. If no value is specified for this parameter, we use a default value of 0 corresponding to the common definition of integration testing (focusing on the current node and using abstractions for all called nodes).

Maximum Path Length. The maximum path length parameter used in the original criteria definition in [6] takes on a slightly new meaning in our criteria definition with the node operators and the integration depth parameter. For instance, in the node of Figure 4, for the paths with length 4, we can identify path $p = \langle in_2, e_2, s_2, out \rangle$ if we consider an integration depth of 0 (i.e. `NODE0` is abstracted using the node operator). However, if we use an integration depth of 1, `NODE0` will be unfolded. The previously considered path $p = \langle in_2, e_2, s_2, out \rangle$ with length 4 will now correspond to $p' = \langle in_2, b, d, out \rangle$ with a path length of 4 or to $p'' = \langle in_2, b, d, t_1, d, out \rangle$ with a path length of 6 if we consider one cycle. Additionally, the computation of the abstract activation conditions associated with a `NODE` operator for called node N depends on the maximum path length and number of cycles we choose in N.

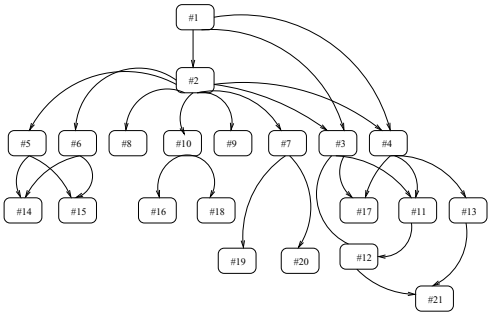


Fig. 5. Structure of a large application

Criteria Definition. We extend the family of criteria defined for unit testing in Section 2.2 (BC, ECC, MCC), for integration testing with the `NODE` operator. We name the extended criteria *iBC*, *iECC,* and *iMCC* (*i* stands for "integration-oriented").

Let m be the integration depth, n the maximum path length, P_n the set of all paths of length lower or equal to n at this depth, l the maximum path length to be considered for the abstract activation condition computation in NODE operators, and $\mathcal{T}$ the set of input sequences. Let $in\,(p)$ denote the input of path p and e denote an internal edge.

The integration-oriented Basic Coverage criterion (*iBC*) requires activating at least once all the paths in the set P_n for the given depth of integration.

Definition 2. *The operator network is covered according to the integration-oriented basic coverage criterion* $iBC^m_{n,l}$ *if and only if:* $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $AC^m_l\,(p) = true$.

The integration-oriented Elementary Conditions Coverage criterion (*iECC*) requires activating each path in P_n for both possible values of its boolean inputs, *true* and *false*.

Definition 3. *The operator network is covered according to the elementary conditions criterion* $iECC^m_{n,l}$ *if and only if:* $\forall p \in P_n$, $\exists t_1 \in \mathcal{T}$: $in\,(p) \wedge AC^m_l\,(p) = true$ *and* $\exists t_2 \in \mathcal{T}$: $not\,(in\,(p)) \wedge AC^m_l\,(p) = true$.

The integration-oriented Multiple Conditions Coverage criterion (*iMCC*) requires paths in P_n to be activated for every value of all its boolean edges, including internal ones.

Definition 4. *The operator network is covered according to the integration-oriented multiple conditions criterion* $iMCC^m_{n,l}$ *if and only if:* $\forall p \in P_n$, $\forall e \in p$, $\exists t_1 \in \mathcal{T}$: $e \wedge AC^m_l\,(p) = true$ *and* $\exists t_2 \in \mathcal{T}$: $not\,(e) \wedge AC^m_l\,(p) = true$.

Subsumption Relation. From the above definitions, it clearly follows that the satisfaction of a criterion $iC \in \{iBC, iECC, iMCC\}$ for a maximum path length n in an operator network implies the satisfaction of the criterion for all path lengths less than n in the operator network. This subsumption relation

is assuming a given integration depth m and maximum path length l for the abstract activation conditions. In other words, $iC^m_{s,l} \subseteq iC^m_{n,l}$ for any $s \leq n$.

With regard to the integration depth parameter, there is no subsumption relation. Consider, for instance, $iBC^m_{n,l}$ and $iBC^{m-1}_{n,l}$. The satisfaction of the former requires to unfold some node calls that are abstracted in the latter at depth *m-1*. As a result, the paths of length less than or equal to n may be different between the two criteria and, therefore, the subsumption relation is not obvious. For the same reason, the original criteria definitions do *not subsume* the extended criteria. Abstraction using the node operator results in a different operator network for the extended criteria from the original criteria. As a result, for the same maximum path length n in the operator network of the global node, the set of paths considered for satisfaction of criterion $C \in \{BC, ECC, MCC\}$ does not necessarily subsume the set of paths considered for satisfaction of criterion $iC \in \{iBC, iECC, iMCC\}$.

Lustructu Tool[3]**.** For coverage measurement using the extended criteria definitions presented in this Section, we enhanced the existing Lustructu tool presented in [6] to support abstraction of node calls and computation of abstract activation conditions. We currently only support an integration depth of 0, i.e., all node calls are abstracted using the *NODE* operator. In the future, we plan to support other values of the integration depth parameter. The Lustructu tool takes as input the Lustre program, name of the node to be integrated, coverage criteria to be measured, and the parameters for the coverage criteria (path length, number of cycles). The tool computes and returns the activation conditions for the integrated node for the selected criteria.

4 Empirical Evaluation

In this section, we evaluate the relative effectiveness of the extended coverage criteria against the original Lustre coverage criteria using an Alarm Management system (AMS) that was developed for an embedded software in the field of avionics. The main functionality of the system is to set off an alarm when the difference between a flight parameter provided by the pilot, and the value calculated by the system using sensors exceeds a threshold value. We believe the system, even though relatively small, is representative of systems in this application area. The AMS is implemented using twenty one Lustre nodes and has several calls between nodes.

Table 3 provides size information on the biggest seven of the twenty one nodes in the system[4]. The main node in the system (node #1) calls node #2 and node #3 that together implement the core functionality of the system. For our evaluation, we use node #2 in the AMS, since it contains several nested temporal loops and two levels of integration, rendering it complex and interesting for us. The two levels of integration in node#2, as depicted by the call graph in Figure 6,

[3] `http://membres-liglab.imag.fr/parissis/TOOLS/LUSTRUCTU/`

[4] The remaining fourteen nodes are relatively small and not complex.

Table 3. Size of the alarm management system nodes

	Code			Op. Net.	
node	*LOC*	*inputs*	*outputs*	*edges*	*operators*
#1	830	29	13	275	190
#2	148	10	3	52	32
#3	157	10	1	59	37

	Code			Op. Net.	
node	*LOC*	*inputs*	*outputs*	*edges*	*operators*
#4	148	10	3	52	32
#5	132	6	5	36	24
#6	98	9	2	33	22
#7	96	7	2	33	22

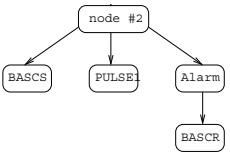


Fig. 6. Call graph for node #1

is as a result of calls to three nodes, two of which in turn call another node. In our evaluation of node #2, we attempt to answer the following two questions,

1. Does the proposed integration oriented criteria reduce the *testing effort* when compared to the original criteria for node integration?
2. Is the proposed integration oriented criteria effective in *fault finding*?

To answer the first question we observe (1) whether the number of activation conditions needed to satisfy the extended criteria is lower than what is needed for the original criteria, and (2) whether test sequences needed for achieving coverage using the extended criteria are shorter than test sequences needed for the original criteria. To address the second question with regard to fault finding effectiveness, we create several mutations of node#2 and determine whether tests are effective in revealing the mutants.

4.1 Testing Effort

In this section we attempt to answer the first question in our evaluation with regard to testing effort. We assess the testing effort in two ways (1) using number of activation conditions, and (2) using test sequence length.

Number of Activation Conditions. Comparing the number of activation conditions between the two criteria helps assess the relative difficulty in satisfying the criteria. It also serves as a good indicator of the relative effort that needs to be spent in coverage measurement and analysis, since more activation conditions usually means a greater number of test cases and longer test case execution time. Analysis of why satisfactory coverage is not achieved becomes difficult with a large number of activation conditions. In our evaluation, we compare the number of activation conditions required for iBC vs BC, iECC vs ECC, and iMCC vs MCC to assess the relative testing effort involved.

Table 4. Number of activation conditions for node #2

	# ACs					
# cycles	iBC	iECC	iMCC	BC	ECC	MCC
1	29	58	342	50	100	1330
3	29	58	342	131	262	4974

In Table 4, we present the number of activation conditions for the extended versus the original criteria for node #2. The data in the table corresponds to complete paths with at most 1 and 3 cycles. As seen in Table 4, the extended criteria, iBC, iECC and iMCC, require significantly fewer activation conditions than their counterpart with no integration, BC, ECC, and MCC, respectively[5]. The difference in number of activation conditions is particularly large between the iMCC and MCC criteria, 342 vs 1330 for 1 cycle and 342 vs 4974 for 3 cycles. As the number of cycles increase, the difference in activation conditions becomes more considerable. From these results, we conclude that the number of activation conditions for the extended criteria is significantly smaller than the original Lustre coverage criteria for node#2.

Test Sequence Length. We now compare the length of test sequences needed to satisfy the extended criteria versus the original criteria. We use the length of test sequences as an indicator of testing effort based on the assumption that developers would need to spend more time and effort constructing longer test sequences. In order to assess the testing effort in an unbiased fashion, we use randomly generated test sequences and measure both criteria. We generated test sequences varying from length 1 up to 1000. We generated five such sets. We do this to reduce the probability of skewing the results by accidentally picking an excellent (or terrible) set of test sequences. We measured the coverage criteria (BC, ECC, and MCC – with and without node integration) over varying test sequence lengths using the following steps:

1. Randomly generate test input sequences of length 1 upto 1000. Generate five such sets with different seeds for random generation.
2. Using each of the test sequences in step 1, execute the coverage node for the criteria and compute the coverage achieved.
3. Average the coverage achieved over the five sets of test sequences.

Figure 7 shows the relation between coverage achieved and test sequence length for iBC vs BC, iECC vs ECC, iMCC vs MCC. Note that these results consider paths with cycles repeating at most 3 times in the node. Figure 7 illustrates that

[5] Node #2 does not contain any temporal loops at the top level (level 1 in Figure 6). As a result, the number of paths for the extended version with the NODE operator is constant regardless of the number of cycles considered. In contrast, in the original version, the number of paths dramatically increases with the number of cycles since the called nodes contain temporal loops. If Node #2 contained temporal loops at the top level, the difference in activation conditions would be more considerable.

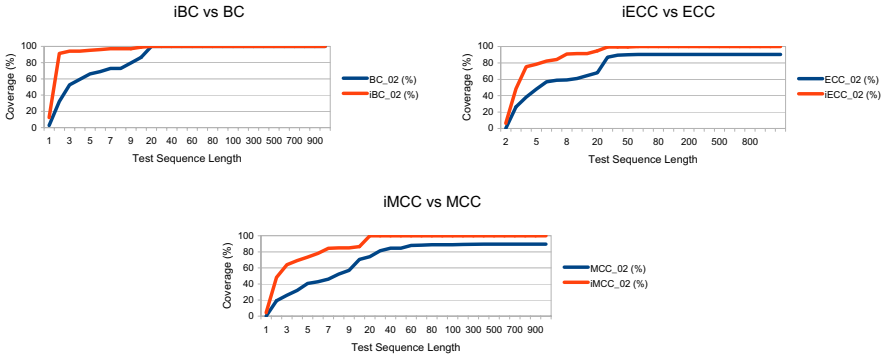


Fig. 7. Comparison of coverage criteria for test sequence lengths 1 to 1000

to achieve the same level of coverage, the extended criteria require shorter test sequences in all three cases - BC, ECC, and MCC. For instance, to achieve 80% iMCC we need test sequences of length 6, whereas to achieve 80% MCC we need test sequences of length 30. Additionally, for all three extended criteria it was possible to achieve 100% coverage with the randomly generated test sequences. On the other hand, for the original criteria, we could only achieve 100% coverage for BC. The maximum coverage achieved for ECC and MCC was only 90% and 89%, respectively, even with test sequences of length 1000. This may be either because the criteria have activation conditions that are impossible to satisfy, or because some of the activation conditions require input combinations that are rare (absent in the generated test sequences).

For iBC vs BC in Figure 7, 100% coverage is achieved for both criteria using test sequences of length at least 20. However, the gradient of the curves for the two criteria are very different. Test sequences of length less than 20 achieve higher iBC coverage than BC. We make a similar observation for iECC vs ECC. Test sequences of length 50 were sufficient to achieve maximum seen coverage for both iECC and ECC (100% for iECC and 90% for ECC). Any given test sequence achieves higher iECC coverage than ECC. For iMCC vs MCC, we find test sequences of length 50 were sufficient to achieve 100% coverage of the iMCC criterion. On the other hand, we need test sequences of length at least 500 to achieve the maximum seen coverage of 89.67% for MCC. In addition, like for BC and ECC, any given test sequence achieves higher iMCC coverage than MCC.

To summarize, the observations made with Figure 7 indicate that the extended criteria require shorter test sequences, and therefore lesser testing effort, than the original criteria for the same level of coverage. The difference is especially significant in the case of iMCC vs MCC. Additionally, it is possible to achieve 100% coverage of the extended criteria but not of the original criteria (except for BC) with the randomly generated test sequences.

Based on these observations and the conclusions about the number of activation conditions made previously, we believe that the extended criteria are more practical and feasible for test adequacy measurement over large systems.

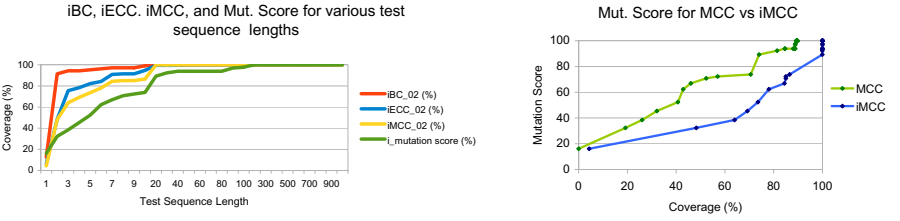


Fig. 8. Mutation Score for Extended Criteria

4.2 Fault Finding Effectiveness

Mutation testing [3] is widely used as a means for assessing the capability of a test set in revealing faults. In our evaluation, we created mutants by seeding a single fault at a time in the original Lustre specification. To seed a fault, we used a tool that randomly selects a LUSTRE operator in the original program and replaces it with a mutant operator in such a way that the mutant program is syntactically correct. Table 5 illustrates the LUSTRE operators we selected and the corresponding mutations for it. We created 26 mutants of node #2. Our procedure for evaluating fault finding effectiveness involved the following steps:

1. Run each of the randomly generated test sequences in Section 4.1 (5 sets of test sequences with length 1 to 1000) on the original Lustre specification of node #2 and record the output.
2. For each of the 26 mutants, run each of the randomly generated test sequences used in step 1 and record the output.
3. For each test sequence, compare the mutant output in step 2 with the corresponding oracle output in step 1. If there is a difference, then the test sequence killed (or revealed) the mutation. Otherwise, the test sequence failed to reveal the seeded fault.
4. The mutation score of a test sequence is the ratio of the number of killed mutants to the total number of mutants (26 in our evaluation).
5. We average the mutation score for each test sequence length over the 5 generated sets.

Figure 8 plots the achieved coverage for iBC, iECC and iMCC and the mutation score. There is a correlation between the criteria satisfaction ratio and the number of killed mutants. This correlation is low for iBC (correlation coefficient of 0.64) since it is a rather weak measure of coverage. Correlation for iECC (0.87) is higher and it is more effective in killing the mutants. iMCC is the most effective in killing the mutants with the highest correlation (0.94). Figure 8 compares

Table 5. Mutations

OPERATOR	NOT	AND	OR	PRE	$<, >, =, \leq, \geq$	$+, -, \star, /$
MUTANT	PRE, $[delete]$	OR, FBY	AND, FBY	NOT, $[delete]$	$<, >, =, \leq, \geq$	$+, -, \star, /$

the mutation score achieved by the extended criterion, iMCC, versus MCC. It is evident that the original criterion is definitely more rigorous than the extended criterion. For instance, achieving 80% MCC reveals 90% of the seeded faults. On the other hand, achieving 80% iMCC only reveals 65% of the seeded faults. Nevertheless, as seen in Section 4.1, achieving the original criterion requires test sequences significantly longer than what is needed for the extended criterion. The shortest test sequence that achieves maximum MCC (89%) is of length 500 and reveals 100% of the faults. The shortest test sequence that achieves maximum iMCC (100%) is of length 50 and reveals 94% of the faults. On an average, this shortest test sequence failed to kill 1 of the 26 mutants. In all 5 sets of randomly generated tests, the shortest test sequence achieving 100% iMCC failed to kill one particular mutant where the `or` LUSTRE operator was replaced by the `fby` operator. Test sequences of length 200 or greater were able to kill this particular mutant. Admittedly, abstraction of the activation conditions over the called nodes leads to a trade off between the test sequence length needed for maximum coverage and the fault finding effectiveness. Nevertheless, as seen in our preliminary evaluation, the iMCC criteria is 94% effective in fault finding, only missing one of the 26 faults, with a reasonably short test sequence.

4.3 Threats to Validity

We face two threats to the validity of our evaluation. First, the AMS is relatively small when compared to other industrial systems. However, it is representative of other systems in the embedded systems domain. Second, the mutants were created by changing the LUSTRE operators. These mutants may not be representative of faults encountered in practice. However, since information on faults that commonly occur is not easy to come by, the fault finding assessment using the mutations is a helpful indicator of the effectiveness of the technique.

5 Conclusion

Previous work [6] proposed structural coverage criteria over LUSTRE programs for measuring unit testing adequacy. This criteria when applied to integration testing experiences severe scalability issues because of the exponential increase in the number of activation conditions to be covered for nodes with calls to other nodes. In this paper, we have presented an extended definition of the LUSTRE structural coverage criteria, called iBC, iECC, iMCC, that helps to address this scalability problem. We use an abstraction that replaces the calls to nodes with a *NODE* operator. This abstraction avoids the entire set of paths of the called node from being taken into account for coverage assessment. However, the abstraction ensures that we can still determine whether the output depends on a given input of the called node. To provide flexibility in dealing with specific needs and complexity of systems, we provide parameters in the criteria definition such as integration depth and path length inside integrated nodes that can be tuned by developers. We hypothesize that the extended criteria will result in considerable

savings in testing effort while still being effective at fault finding. We conducted a preliminary evaluation of this hypothesis using an Alarm Management System. The extended criteria reduced the number of activation conditions by as much as 93% (for MCC) and was effective at revealing 94% of the seeded faults.

References

1. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
2. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing 9(5), 193–200 (1994)
3. DeMillo, R.A.: Test Adequacy and Program Mutation. In: International Conference on Software Engineering, pp. 355–356. Pittsburg, PA (1989)
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
5. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language lustre. IEEE Trans. Software Eng. 18(9), 785–793 (1992)
6. Lakehal, A., Parissis, I.: Structural coverage criteria for lustre/scade programs. Softw. Test., Verif. Reliab. 19(2), 133–154 (2009)
7. Marre, B., Arnould, A.: Test sequences generation from lustre descriptions: Gatel. In: IEEE International Conference on Automated Software Engineering, Grenoble, France, pp. 229–237 (October 2000)
8. Papailiopoulou, V., Madani, L., du Bousquet, L., Parissis, I.: Extending structural test coverage criteria for lustre programs with multi-clock operators. In: The 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), L'Aquila, Italy (September 2008)
9. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: IEEE Real-Time Systems Symposium, pp. 200–209 (1998)
10. Woodward, M.R., Hedley, D., Hennell, M.A.: Experience with path analysis and testing of programs. IEEE Trans. Softw. Eng. 6(3), 278–286 (1980)

Formal Analysis of a Triplex Sensor Voter in an Industrial Context

Michael Dierkes

Rockwell Collins France, 6 avenue Didier Daurat, 31701 Blagnac, France
mdierkes@rockwellcollins.com

Abstract. For several years, Rockwell Collins has been developing and using a verification framework for MATLAB Simulink© and SCADE Suite™models that can generate input for different proof engines. Recently, we have used this framework to analyze aerospace domain models containing arithmetic computations. In particular, we investigated the properties of a triplex sensor voter, which is a redundancy management unit implemented using linear arithmetic operations as well as conditional expressions (such as saturation). The objective of this analysis was to analyze functional and non-functional properties, but also to parameterize certain parts of the model based on the analysis results of other parts. In this article, we focus on results about the reachable state space of the voter, which prove the bounded-input bounded-output stability of the system, and the absence of arithmetic overflows. We also consider implementations using floating point arithmetic.

1 Introduction

In order to meet their reliability requirements, safety-critical systems such as digital flight control typically rely on redundant hardware. Redundancy management [8] has been investigated since the 1960s. While first implementations were based on analog hardware, the introduction of digital computers in the 1970's enabled the development of systems based on fault detection and isolation. A common form of redundancy used in aircraft systems is Triple Modular Redundancy (TMR), in which three instances of a device run in parallel and a voter is used to process the results of these instances.

TMR can be applied on different levels of a system, starting with the devices that furnish critical information about its environment and therefore are required to provide high reliability. Sensors that measure physical quantities like air data on aircraft are often exposed to extreme conditions and may be subject to temporary or even permanent faults. To increase the integrity of the measurement, TMR is implemented in this context by using three identical sensors and computing an output value from the three input values by a triplex sensor voter algorithm, that might for example calculate an average, or select the midvalue. If one of the sensor values exhibits an unacceptable discrepancy from the two other values, it is considered as faulty, and the voter switches to a degraded mode in which only the two valid inputs are taken into account. This implies

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 102–116, 2011.

that TMR is based on the assumption that two sensors never fail exactly at the same instant, which must be justified by the very low probability of such an event.

A sensor can be considered as defective if the difference with the other sensors is very high over a short time span, or if it is moderately different over a longer period of time. The thresholds that are used in the fault detection have to be chosen carefully since a sensor that is still in the limits of its tolerance should not be disconnected because a threshold was chosen too low. Conversely, a threshold chosen too large may compromise the measurement in a potentially dangerous way. Furthermore, when a sensor is disconnected, the transient of the voter output must be limited so as to not perturb the systems that rely on the voter output.

The problems of choosing the threshold values and guaranteeing an upper bound for the transient can benefit from a formal analysis that can safely approximate all possible internal states of a system and thereby consider all possible input scenarios. In this way, it can be guaranteed that the sensor voter design meets its safety requirements. Usually, the fault detection thresholds and transients are determined by simulation, which is very time consuming since many different cases need to be considered, and does not guarantee that input sequences that lead to extreme cases have been taken into account. Furthermore, even if the algorithm of a system exhibits a correct behaviour on the model level, it is still possible that an implementation behaves incorrectly due to the use of floating point hardware in which the occurrence of rounding errors is unavoidable in general. In this article, we describe our experience with using formal analysis to address these issues. We do not present any particular new methodology, but we investigate if and how existing analysis tools can be used to satisfy industrial needs.

An additional objective of this article is to make the triplex sensor voter publicly available to the research community, who might use it as an industrial case study in order to experiment new techniques. Both model checking and abstract interpretation could be used to achieve a higher degree of automatization and a more precise representation of the reachable state space. Following this objective, some of the results described here have been presented orally at workshops in the past (like for example [4]), however they have not been the subject of a refereed written article.

The rest of this article is organized as follows. In Section 2, we describe the system that we have analyzed, a triplex sensor voter, and in Section 3, we present the objectives and results of an analysis of the Simulink model which specifies the voter. Implementations using floating point arithmetic are considered in Section 4, where we present analysis results about such implementations. In Section 5, we describe a simple method to find invariants automatically, which showed to be very useful in practice. The lessons we have learned from the analysis are presented in Section 6, and we conclude with an outlook on future work in Section 7.

2 The Triplex Voter

The triplex sensor voter we analyzed is an example for a voter that is used on commercial aircraft. It is modeled in the widely used Simulink [10] environment, and we used the Rockwell Collins translation framework [7] to analyze it using formal methods, much as was done in the work presented in [3]. However, in [3], real values are abstracted by integer values, and the analyzed voter is different in that it is implemented by a pure function, in contrast to the voter in the present work which uses integrators.

The triplex sensor voter computes an output value from inputs provided by three redundant sensors, possibly detecting a sensor failure when an input value presents a sufficiently large mismatch with respect to the other inputs over a certain period of time. In its Simulink model it is possible to distinguish blocks that implement the control part (reset logic, failure processing) from blocks that implement the output value computation. The former consists mainly of operations with Boolean output (comparators, logical operations) while the latter contains arithmetic operations. Following a compositional verification approach, we consider these blocks separately. In the work described in this article we focused on the output value computation block and will not describe the failure detection logic here. In its normal operational mode in which no failure has been detected, the voter takes three input values and determines the output value in the following way: for every input value there is an internal *equalization* value that is subtracted from the corresponding input value in order to obtain an *equalized* value. Then, the middle value of the three equalized values is chosen. A block diagram of the voter without failure detection is shown in figure 1.

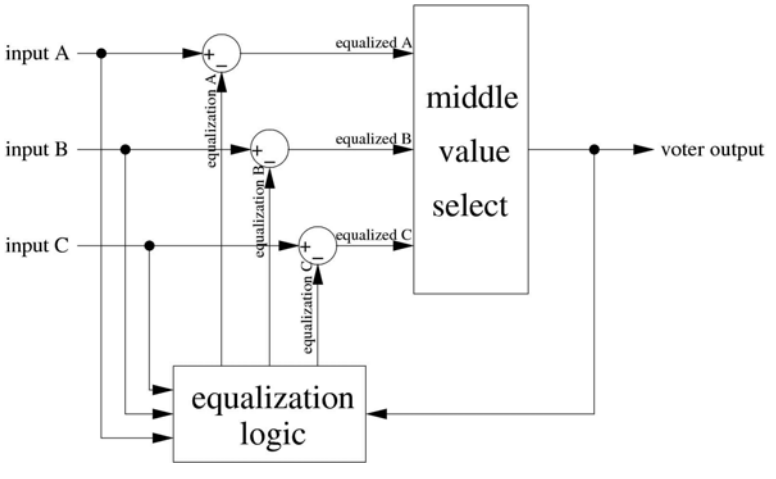


Fig. 1. Block diagram of the triplex sensor voter (without failure detection)

Note that the voter does not compute an average value from the three input values, but chooses the middle value of the three equalized values. This has the advantage of robustness against failures in which one of the sensors gives suddenly extremely high or extremely low values since the output does not depend on the maximal and the minimal values, but only on the middle value.

The role of the equalization values is to compensate offset errors of the sensors, assuming that the middle value gives the most accurate measurement. For example, if the input values have the form $(x_t + 1.0, x_t, x_t - 1.0)$, the equalization values will tend to $(1.0, 0.0, -1.0)$, thus the equalized values tend to (x_t, x_t, x_t). The behaviour of the voter is illustrated by the diagram in figure 2. Since the subject of this article is the analysis of the voter, but not the voter itself, we will not give any further explanation or motivation of its behaviour. It might even be preferable to have an analysis method which does not require a deeper understanding of the dynamical behaviour of the system itself.

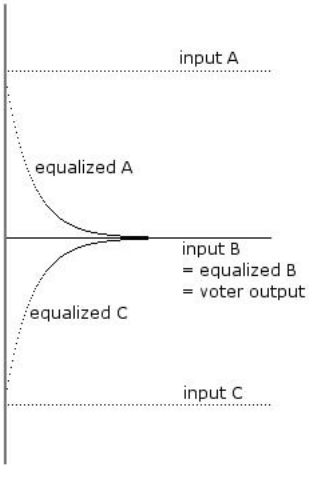


Fig. 2. Simulation trace of the triplex voter: if the input values stay constant, the equalized values tend to the middle input value. The middle equalized value is chosen as output.

Let $sat_l(x)$ denote saturation of x to l, i.e.

$$sat_l(x) = \begin{cases} l & \text{if } x > l \\ -l & \text{if } x < -l \\ x & \text{otherwise} \end{cases}$$

The behaviour of the voter in normal operation mode can then be described by the following recurrence relations:

$$
\begin{aligned}
EqualizationA_0 &= 0.0 \\
EqualizationB_0 &= 0.0 \\
EqualizationC_0 &= 0.0
\end{aligned}
$$

$$
Centering_t = middleValue(EqualizationA_t, EqualizationB_t, EqualizationC_t)
$$

$$
\begin{aligned}
EqualizedA_t &= InputA_t - EqualizationA_t \\
EqualizedB_t &= InputB_t - EqualizationB_t \\
EqualizedC_t &= InputC_t - EqualizationC_t
\end{aligned}
$$

$$
VoterOutput_t = middleValue(EqualizedA_t, EqualizedB_t, EqualizedC_t)
$$

$$
\begin{aligned}
EqualizationA_{t+1} &= EqualizationA_t + \\
&\quad 0.05 * (sat_{0.5}(EqualizedA_t - VoterOutput_t) - sat_{0.25}(Centering_t)) \\
EqualizationB_{t+1} &= EqualizationB_t + \\
&\quad 0.05 * (sat_{0.5}(EqualizedB_t - VoterOutput_t) - sat_{0.25}(Centering_t)) \\
EqualizationC_{t+1} &= EqualizationC_t + \\
&\quad 0.05 * (sat_{0.5}(EqualizedC_t - VoterOutput_t) - sat_{0.25}(Centering_t))
\end{aligned}
$$

These equations can easily be expressed in a synchronous language like Lustre [2] in order to analyze them with formal verification tools. In practice, a Lustre description can be obtained automatically from the original Simulink model by using the Rockwell Collins verification framework [7].

Note that the state of the voter at a given time step is completely determined by the three equalization values: the output is computed from the input values and from the equalization values.

The saturation parameters 0.5 and 0.25 have been fixed exemplarily. They limit the maximum rate of change of the voter output and depend on the context in which the voter is used. The parameter 0.05 expresses the sensor sample rate since the voter is usually run with a frequency of 20Hz. In section 5, we will mention how we have analyzed the influence of these parameters.

3 Analysis of the Simulink Model

Analysis of the triplex voter was performed using both formal verification and simulation for the case in which no sensor can fail. Additional analysis is being performed for the more complicated case in which at most one sensor can fail. For the formal verification, we used the Prover [9] and the Kind [5] inductive provers. When we considered only the equations corresponding to normal operation, we mainly used Kind which takes advantage of the SMT solving capabilities of the underlying solver. However, on the full model (including fault detection), Kind was not able to prove the investigated properties, whereas Prover furnished good results thanks to its capabilities for automatic invariant generation.

3.1 Properties of Interest

The first analysis objective we were interested in was to prove the bounded-input bounded-output stability of the system, i.e. that the system output is bounded as long as the system input is bounded, so that no transient peaks can occur which might perturbate the control systems which receive the voter output as their input. In particular, we wanted to prove that the maximal difference between the voter output and the true value of the quantity measured by the sensors (represented by the variable $TrueValue$) is bounded. We assume that all three input values differ by at most a certain constant value $MaxDev$ from $TrueValue$, that corresponds to the maximal deviation of a sensor when it operates normally. This is formally expressed by the constraints

$$\begin{aligned} |InputA - TrueValue| &\leq MaxDev \\ |InputB - TrueValue| &\leq MaxDev \\ |InputC - TrueValue| &\leq MaxDev \end{aligned}$$

In order to prove that the output value is bounded, it is sufficient to prove that the centering value is bounded since we can easily prove that for a given constant C,

$$|Centering| < C \;\rightarrow\; |VoterOutput - TrueValue| < MaxDev + C$$

Since the equalization values depend only on the differences between the input values and not on their absolute value, we can assume without loss of generality that the true value is 0.0 to simplify the analysis.

We were also interested in relations between the equalization values that hold over all reachable states, for example the maximal difference between any two equalization values. Fault detection (which is not described in this article) is based on monitoring the equalization values and the equalized values. In the more complex case in which a sensor can fail, analysis of the equalization values should allow us to parameterize the fault detection logic and prove that it behaves correctly.

Furthermore, we were interested to prove an upper bound for the equalization values, which implies that no arithmetic overflow may occur even if the voter is operating over long periods of time.

3.2 Analysis Results

In our analysis, we fixed the maximal sensor deviation to 0.2. This value was recommended as typical by the domain experts and was also large enough to ensure that the conditional logic implemented in the saturation blocks was invoked. By running simulation, we found a possible maximal value of about 0.151 for the centering value. We also tried bounded model checking in order to find a lower bound for the maximal difference between the true value and the voter output. However, due to the coefficient 0.05 in the definition of the equalization values, the change rate of these values is very small and it takes a large number

of steps to reach extreme values. With bounded model checking, we only found a maximal centering value of about 0.1.

We also tried an iterative approach, where we first generated a counter example for $|Centering| < C_1$ for a small value C_1. Then, in the next iteration we took the state violating $|Centering| < C_1$ found in the previous iteration and used it as initial state to find a counterexample for $|Centering| < C_2$, with $C_2 > C_1$, and so on for $C_3, C_4, \ldots$ until a value was reached for which no counterexample could be found. This gave slightly better results, but the value found was still smaller than the one found by simulation, which is certainly due to the analysis being stuck at a local maximum.

Using the SMT solvers Kind and Prover, we found that the conjunction of the following expressions constitutes a 1-inductive invariant over the triplex voter state:

$$
\begin{gathered}
|EqualizationA| < 0.4 \\
|EqualizationB| < 0.4 \\
|EqualizationC| < 0.4 \\
|EqualizationA + EqualizationB + EqualizationC| < 2/3 \\
|EqualizationA - EqualizationB| < 0.4 \\
|EqualizationB - EqualizationC| < 0.4 \\
|EqualizationC - EqualizationA| < 0.4 \\
|Centering| < 0.27
\end{gathered}
$$

This invariant was found by hand, using the trial and error principle. In the course of proving these invariants, we also considered and proved several other expressions, but these did not turn out to be useful in proving our main theorems. Finding these invariants through trial and error was very time consuming, even if the execution time of the model checker was never more than a few seconds. Note that the invariant we have found implies a maximal centering value of 0.27. By increasing the number of induction steps to 7, we where able to prove that 0.24 is an upper bound for the centering value. There is still uncertainty for the interval between the maximal value found by simulation (slightly less than 0.151) and 0.24.

The first four lines of the above inequalities already define an inductive invariant. However, the other inequalities are interesting because they give an upper bound for the difference of two equalization values, which is useful for parameterizing the fault detection logic, and an upper bound for the centering value, which implies an upper bound on the difference between the voter output and the true value.

The validity of this invariant can be checked by a pencil-and-paper approach which is tedious, but the symmetry of the voter can be used to limit the number of cases that need to be considered. We are not aware of any simple mathematical argument which could explain that the equalization values stay within the bounds given by the invariant, and which could be found easily by the designer of the system, thus there is a real interest in a computerized formal analysis.

To improve our insight into why these particular invariants were useful, we also generated system states through random simulation. Figure 3 shows states

that were reached during simulation, as well as the approximation of the reachable state space by the inductive invariant. More precisely, it shows a projection of these states to the (EqualizationA, EqualizationB)-plane (the state space itself is 3-dimensional and includes EqualizationC). The two-dimensional projections of states reached by simulation are represented as black points while the approximation of the reachable state space defined by the inductive invariant is shown as a shaded polygon. In the simulation, all input values were extreme values, i.e. equal to $\pm MaxDev$, and changed after every 150 steps. It is likely (however not guaranteed) that certain points that were reached are at the outer bounds of the state space. From the diagram, we might conclude that the space of reachable states has a relatively complex form, and cannot be described by linear expressions in a precise manner. The simulation results shown here do not claim to be complete in any way, but they are only supposed to give an idea about the system behaviour, and about maximal values which can effectively be reached during an execution.

In order to prove an upper bound for *Centering* that is smaller than 0.24, the approximation needs to be more precise. However, if we diminish any of the constants on the right hand side of the inequalities, then the invariant is no longer inductive.

Note that the square areas on the upper right and lower left corner come from the inequality $|Centering| < 0.24$ since this implies that two equalization values cannot be both greater than 0.24 (otherwise the middle value would be greater than 0.24). The uncertainty that still persists in our analysis would get smaller if either we could make these squares bigger (i.e. prove an upper bound for centering that is less than 0.24) or find a scenario in which the reached states come closer to the squares (centering greater than 0.151).

4 Analysis of a Floating Point Implementation

The results we obtained by analyzing the model are based on a mathematical semantics in which real variables are supposed to have infinite precision, i.e. rounding errors due to a finite representation do not occur. For a machine implementation, which usually uses floating point arithmetic, these results might not be true. For example, we cannot exclude that the accumulation of rounding errors could jeopardize the stability of the system after several hours of operation. For this reason, a formal analysis of the pure model is not sufficient, but we also need to analyze the impact of rounding errors.

4.1 Analysis Using an SMT Solver

Similar to the approach presented in [6], we used an upper approximation of the rounding error and an SMT solver to prove the stability of the system, under the assumption that the used compiler does not change the ordering of the operations. Floating point implementations could be analyzed at bit level in a completely accurate way, but this is very costly since floating point arithmetics

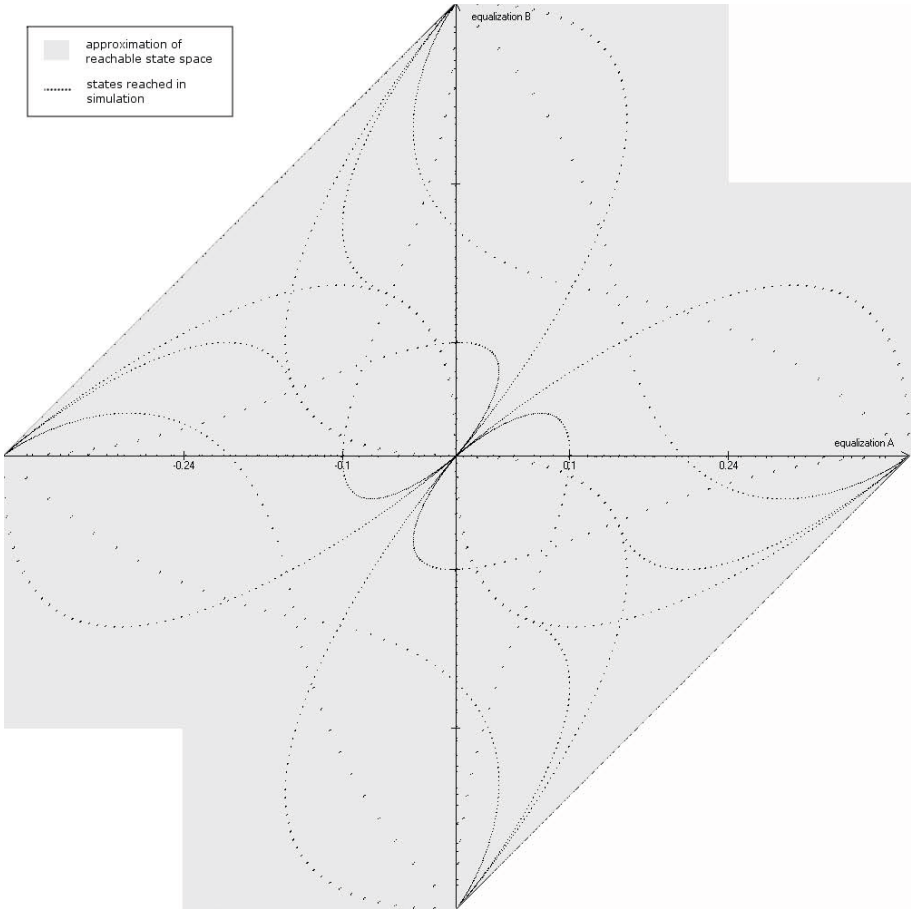


Fig. 3. Approximation of the reachable state space, and execution trace of a simulation (projection on the EqualizedA-EqualizedB-plane)

correspond to complex boolean functions on large bit vectors. Also, all the details about the executing processor and the width of its internal registers need to be known, and we could not take advantage of the capabilities of SMT solvers since the reasoning is exclusively on boolean level. For our purpose, a much less precise analysis is sufficient.

Modern floating point hardware is usually conform to the IEEE 754 standard, in which two kinds of numbers are distinguished: normalized numbers (with a mantissa greater or equal to 1.0 and less than 2.0) and subnormal numbers (with a mantissa less than 1.0 and with the smallest representable exponent). For our analysis, we need to have the following information about the hardware on which the code is executed:

- an upper approximation of the maximal relative error of normalized numbers (which depends on the number of digits in the representation of the mantissa), and
- an upper approximation of the maximal absolute error of subnormal numbers (depending on the smallest representable exponent, and on the number of digits of the mantissa).

Like in [6], in order to keep the model as simple as possible, we use the sum of the maximal relative error and of the maximal absolute error as an upper approximation of the rounding error of floating point operations. The rounding errors themselves are represented by free variables which are introduced into the model, one for every arithmetic operation. Let $relErr$ denote the maximal relative error of normalized numbers, and $absErr$ denote the absolute error of subnormal numbers. For example, if 32 bit IEEE 754 floating point numbers and rounding-to-nearest mode are used, $relErr = 2^{-24}$, and $absErr = 2^{-150}$.

For an arithmetic operation $\otimes \in \{+, -, *, /\}$, let $\otimes_F$ be the floating point operation corresponding to $\otimes$, and let f_1 and f_2 be values which are representable in the used floating point representation. Let

$$E = |f_1 \otimes f_2| * relErr + absErr$$

Then it holds that

$$(f_1 \otimes f_2) - E \leq f_1 \otimes_F f_2 \leq (f_1 \otimes f_2) + E$$

In practice, we replace every expression

$$a \otimes b$$

in the model by

$$a \otimes b + e$$

where e is a new variable which is constrained by

$$|e| \leq |a \otimes b| * relErr + absErr$$

For numerical constants which occur in the model, we can determine the precise rounding error, and thus replace constants by their floating point approximation. For example, if 32 bit IEEE 754 floating point numbers are used, the constant 0.05 is replaced by 0.0500000008.

After this modification of the model, the inequalities we gave in the preceding section do not define an invariant anymore. However, we can find an invariant which is slightly larger than the one for real number semantics. For a maximal sensor deviation of 0.2, the conjunction of the following expressions defines an inductive invariant:

$$
\begin{aligned}
|EqualizationA| &< 0.4 + 1.0\text{E-}6 \\
|EqualizationB| &< 0.4 + 1.0\text{E-}6 \\
|EqualizationC| &< 0.4 + 1.0\text{E-}6 \\
|EqualizationA + EqualizationB + EqualizationC| &< 2/3 + 2.0\text{E-}6
\end{aligned}
$$

We tried to use more precise approximations of the rounding error, for example taking into account that the absolute rounding error is zero for addition and subtraction, or that the rounding error is smaller than both operands. However, we were not able to find a smaller invariant, or even to prove the invariant of the infinite precision semantics. From a practical point of view, the invariant that we found is clearly sufficient.

4.2 Analysis by Abstract Interpretation

Abstract interpretation is a powerful technique to check properties on the source code level. Astrée [1] is a tool based on abstract interpretation that has been used with success on many industrial applications to prove the absence of runtime errors, where rounding errors are taken into account. We have used Astrée on a manually written C code implementation of the voter.

In its standard configuration, Astrée did not find an invariant, and reported a potential overflow of the equalization values. However, Astrée was able to confirm partially the invariant that we have found as described in the preceding section, i.e. it was able to prove that certain transitions of the voter do not violate the invariant. We believe that in principle, it should be possible to confirm the invariant for all transitions if the code is modified in a way which allows Astrée to exploit all available information. However, this requires a high effort since all possible execution paths have to be considered separately. Future versions of Astrée might not require this transformation.

A possible interaction between model level and code level would be to prove invariants on model level using an SMT solver and to transfer the invariant to code level where it is confirmed using an abstract interpretation tool. Such proofs could be used as certification evidence in the future since a qualification support kit for Astrée is planned.

5 Automated Generation of Invariants

Finding inductive invariants by hand is very time consuming and depends on the skills of the user, therefore we were looking for a way to generate them automatically. In the case of the voter analysis, a relatively simple algorithm turned out to be very useful. It allowed us to analyze the effect of varying parameters, and to express invariants depending on the maximal deviation of the sensors and on other system parameters.

Our objective was to find inductive invariants that can be described by a conjunction of the form

$$\bigwedge_{i=1,\ldots,n} |expr_i| \leq C_i$$

where the $expr_i$ are linear expressions over the state variables, and the C_i are non-negative constants. The choice of the expressions is primordial in order to find an invariant. Since we do not have a method to generate the expressions automatically, we can only use a trial-and-error approach. Given appropriate expressions, the following algorithm searches for an invariant:

Input: a set of expressions $expr_1, \ldots, expr_n$,
a Lustre model M,
a constant real number δ
real $v_1, \ldots, v_n := 0.0$
bool $invariant\_found :=$ false
repeat
  if $\bigwedge_{i=1,\ldots,n} |expr_i| \leq v_i$ is a 1-inductive invariant of M **then**
    $invariant\_found :=$ true
  else
    Let Γ be an induction step counter example
    forall $i \in \{1, \ldots, n\}$ **do**
      if $|\Gamma(expr_i)| > v_i$ **then**
        $v_i := v_i + \delta$
until $invariant\_found$;

Algorithm 1. Invariant Generation

Note that we do not increase the value of the variables v_i to the value which has been found in the counter example, but by a constant δ. This is because in a counter example, the values which are found are greater than the values in the tested property only by a very small amount. SMT solvers in general do not generate an optimal solution, and even if we would compute an optimal solution, the convergence would be extremely slow. Worse, the algorithm might not terminate at all. Of course, a bad choice of δ might also lead to a very long execution time if it is chosen too small, or to non-termination if it is chosen too great. We obtained good results with $\delta = 0.01$.

By varying the maximal deviation and observing the effect on the automatically generated invariant, we were able to find a general formulation of the invariant depending on the maximal deviation. For every $MaxDev > 0$, we can prove that the conjunctions of the following expression is an invariant of the voter:

$$|EqualizationA| < 2 * MaxDev$$
$$|EqualizationB| < 2 * MaxDev$$
$$|EqualizationC| < 2 * MaxDev$$
$$|EqualizationA + EqualizationB + EqualizationC| < 2 * MaxDev + 1/3$$
$$|Centering| < MaxDev + 1/8$$

Modifications of the saturation parameters (which we have set to 0.25 and 0.5) do not have any impact on this invariant, with the restriction that the saturation threshold applied to the centering value must be greater or equal to the other threshold divided by two, otherwise the expression is not inductive. Concerning the sensor sample rate (fixed to 0.05), if it is chosen in the interval $]0.0, 0.5]$, the expression is inductive.

Note that for $MaxDev = 0.2$, the above invariant is weaker than the one on page 108. In fact, for values of $MaxDev$ which are smaller than 0.25, the smallest upper bounds which can be proven are less than the values in the general invariant, which is probably due to the non-linearity caused by the saturation operators.

6 Lessons Learned

Our first approach was similar to the analysis of purely boolean systems. In this approach, when an inductive proof fails, the feedback from the proof engine is analyzed by the user, and then missing invariants are added. When a purely Boolean system is analyzed, system invariants often correspond to some implicit intention of the designer. An intuition of how the system is supposed to work can therefore be very helpful to find invariants. However, this approach does not seem to work as well on the kind of system analyzed here. The invariants do not really express an intention of the designer, who is probably not even aware of them. They are implied by the dynamic behaviour of the system, but not a design objective in their own right. Therefore, trying to find invariants by looking at induction step counterexamples is very difficult.

It turned out to be a better approach to consider systematically certain inequalities comparing linear expressions over the system variables (for example, sums or differences of any two variables) to constants and then to try to find the smallest constants for which the set of inequalities is an invariant of the system, which also has the advantage that it can be done automatically.

The runtime of the model checker when we tried to find an inductive invariant was in the order of some seconds, and therefore negligible compared to the time spent by the human users formulating potential invariants and interpreting induction step counter examples.

The proof that a floating point implementation of the voter has bounded-input bounded-output stability and cannot generate runtime errors caused by arithmetic overflow is certainly an interesting result. The approach using SMT-solvers was successful for this. Even if modelizing rounding errors requires the introduction of many new variables, the analysis runtime was in the order of a few minutes.

Abstract interpretation seems to be less suited to this kind of system since the analysis probably needs to consider every execution path separately. This means that it is not possible to apply execution path abstraction, which gives its power to abstract interpretation.

7 Ongoing and Future Work

A gap still exists in our analysis between the greatest centering value that we found by simulation (about 0.151) and the smallest upper bound that we were able to prove (0.24). Therefore, we are interested in either finding a smaller upper bound or counterexample that demonstrates a greater maximal value. In order to prove a smaller upper bound, we need a more precise approximation of the space of reachable states. This could possibly be done using other forms of invariants, like invariants containing boolean conditions, or non-linear invariants (however, the latter would not be accepted by the inductive provers we used). On the other hand, for practical applications a guaranteed upper bound of 0.24 might be sufficient to ensure the correct behaviour of the system the voter is part of.

It seems difficult to obtain a greater maximal centering value by simulation since we have already investigated the most obvious strategy, namely stimulating the system with extreme input values. Another approach would be to develop more powerful techniques based on bounded model checking. Our iterative approach was a first step to this direction.

Our invariant generation procedure requires the user to specify a set of expressions for which upper bounds are searched for. For the moment, there is no assistance in finding these expressions. It would be very interesting to find how these expressions could be derived from the program automatically. In general, a very interesting question is if our invariants, or at least some of them, can be found completely automatically. This question will certainly be investigated further since it is necessary for the analysis to be as automated as possible if it is to be used in an industrial context by engineers who are not experts on formal methods.

Abstract interpretation seemed to be less powerful for proving the invariant of the voter than model checking, but as soon as the full triplex voter including fault detection and reset logic is analysed, model checking is likely to require a huge amount of computing resources because of its lack of scalability. In this case, abstract interpretation might be very useful. The best solution might be to combine both techniques, and a research effort to do this is currently undertaken at Rockwell Collins. Furthermore, we made the experience that abstract

interpretation can benefit from the code being structured in a certain way. Automated generation of code which is optimized for analysis by abstract interpretation should be investigated more in depth.

In our analysis of a floating point implementation, we used a very simple approximation of the rounding error. It would be interesting to study more precise approximations, which might allow to prove the invariant that was found on the model level without modification. Concerning execution time, we made the experience that a more precise approximation sometimes allows for a much faster analysis, despite the additional formulas which are necessary. However, a systematic analysis of this needs to be done.

References

1. Absint Angewandte Informatik GmbH, Astrée product description, `http://www.absint.com/astree`
2. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL, pp. 178–188 (1987)
3. Dajani-Brown, S., Cofer, D.D., Hartmann, G., Pratt, S.: Formal modeling and analysis of an avionics triplex sensor voter. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 34–48. Springer, Heidelberg (2003)
4. Dierkes, M.: Analysis of a triplex sensor voter at Rockwell Collins France. Oral presentation at the TAPAS workshop without article (2010), `http://www.di.ens.fr/tapas2010/TAPAS_Michael_Dierkes.pdf`
5. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: Cimatti, A., Jones, R.B. (eds.) FMCAD, pp. 1–9. IEEE, Los Alamitos (2008)
6. Ivancic, F., Ganai, M.K., Sankaranarayanan, S., Gupta, A.: Numerical stability analysis of floating-point computations using software model checking. In: MEMOCODE, pp. 49–58. IEEE Computer Society, Los Alamitos (2010)
7. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM 53(2), 58–64 (2010)
8. Osder, S.: Practical view of redundancy management application and theory. Journal of Guidance Control and Dynamics 22(1), 12–21 (1999)
9. Prover Technology, Prover plug-in product description, `http://www.prover.com`
10. The Mathworks, Simulink product description, `http://www.mathworks.com`

Experiences with Formal Engineering: Model-Based Specification, Implementation and Testing of a Software Bus at Neopost.*

Marten Sijtema[1], Mariëlle I.A. Stoelinga[2],
Axel Belinfante[2], and Lawrence Marinelli[3]

[1] Sytematic Software, the Hague, The Netherlands
marten@sytematic.nl
[2] Faculty of Computer Science, University of Twente, The Netherlands
{marielle,axel.belinfante}@cs.utwente.nl
[3] Neopost, Austin, Texas, USA
l.marinelli@neopost.com

Abstract. We report on the actual industrial use of formal methods during the development of a software bus. At Neopost Inc., we developed the server component of a software bus, called the *XBus*, using formal methods during the design, validation and testing phase: We modeled our design of the XBus in the process algebra mCRL2, validated the design using the mCRL2-simulator, and fully automatically tested our implementation with the model-based test tool JTorX. This resulted in a well-tested software bus with a maintainable architecture. Writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time. Moreover, the errors found with model-based testing would have been hard to find with conventional test methods. Thus, we show that formal engineering can be feasible, beneficial and cost-effective.

1 Introduction

Formal engineering, that is, the use of formal methods during the design, implementation and testing of software systems is gaining momentum. Various large companies use formal methods as a part of their development cycle; and several papers report on the use of formal methods during ad hoc projects [27,15].

Formal methods include a rich palette of mathematically rigorous modeling, analysis and testing techniques, including formal specification, model checking, theorem proving, extended static checking, run-time verification, and model-based testing. The central claim made by the field of formal methods is that, while it requires an initial investment to develop rigorous models and perform rigorous analysis methods, these pay off in the long run in terms of better, and

* This research has been partially funded by NWO and DFG by grant Dn 63-257 (ROCKS), and by the European Union under FP7-ICT-2007-1 grant 214755 (QUASIMODO).

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 117–133, 2011.

more maintainable code. While experiences with formal engineering have been a success in large and safety-critical projects [24,17,27,29,30], we investigate this claim for a more modest and non-safety-critical project, namely the development of a software bus.

Developing the XBus. In this paper, we report on our experiences with formal methods during the development of the XBus at Neopost Inc. Neopost is one of the largest companies in the world producing supplies and services for the mailing and shipping industry, like franking and mail inserting machines, and the XBus is a software bus that supports communication between mailing devices and software clients. The XBus allows clients to send XML-formatted messages to each other (the X in XBus stands for XML), and also implements a service-discovery mechanism. That is, clients can advertise their provided services and query and subscribe to services provided by others.

We have developed the XBus using the classical V-model [31], see Fig. 2, using formal methods during the design and testing phase. The total running time of this project was 14 weeks.

An important step in the design phase was the creation of a behavioral model of the XBus, written in the process algebra mCRL2 [23,4]. This model pins down the interaction between the XBus and its environment in a mathematically precise way. Performing this modeling activity greatly increased the understanding of the XBus protocol, which made the implementation phase a lot easier.

Testing of the XBus. After implementing the protocol, we tested the implementation, where we distinguished between data- and protocol behaviour. Data behaviour concerns the input/output behaviour of a function. This behaviour is static, the input/output behaviour is independent of the order in which the methods are called. Protocol behaviour relates to the business logic of the system, i.e. the interaction between the XBus and its clients. Here, the order in which protocol messages occur crucially determines the correctness of the protocol. First, data behaviour was tested using unit testing, and all errors found were repaired. Then, protocol behaviour was tested using JTorX (details below), since the purpose of the mCRL2 model was exactly to pin down the protocol behaviour.

With JTorX. We tested the implementation against the mCRL2 model. JTorX [7,3] is a model-based testing tool (partly) developed during the Quasimodo project [6]. It is capable of automatic test generation, execution and evaluation. During the design phase, we already catered for model-based testing, and designed for testability: we took care that at the model boundaries, we could observe meaningful messages. Moreover, we made sure that the boundaries in the mCRL2 model matched the boundaries in the architecture. Also, to use model-driven test technology required us to write an adapter. This is a piece of software that translates the protocol messages from the mCRL2 model into physical messages in the implementation. Again, our design for testability greatly facilitated the development of the adapter.

Our findings. We ran JTorX against the implementation and the mCRL2 model (once configured, JTorX runs completely automatically) and found five subtle bugs that were not discovered using unit testing, since these involved the order in which protocol messages should occur. After repairing these, we ran JTorX several times for more than 24 hours, without finding any more errors.

Since writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time (counting only human working time), we conclude that the formal engineering approach has been very successful: with limited overhead, we have created a reliable software bus with a maintainable architecture. Therefore, as in [19], we clearly show that formal engineering is not only beneficial for large, complex and/or safety-critical systems, but also for more modest projects.

The remainder of this paper is organized as follows. Section 2 provides the context of the XBus implementation project. Then, Section 3 describes the activities involved in each phase of the development of the XBus. Section 4 reflects on the lessons learned in this project and finally, we present conclusions and suggestions for future work in Section 5.

2 Background

2.1 The XBus and Its Context

Neopost Incorporated [5] is one of the world's main manufacturers of equipment and supplies for the mailing industry. Neopost produces both physical machines, like franking and mail inserting machines, as well as software to control these machines. Neopost is a multinational company headquartered in Paris, France that has departments all over the world. Its software division, called Neopost Software & Integrated Solutions (NSIS) is located in Austin, Texas, USA. This is where the XBus implementation project took place.

Shipping and franking mail. Typically, the workflow of shipping and franking is as follows. To send a batch of mail, one first puts the mail into a folding machine, which folds all letters, then an inserting machine inserts all letters into envelopes[1] and finally, the mail goes into a franking machine, which puts appropriate postage on the envelopes and keeps track of the expenses.

Thus, to ship a batch of mail, one has to set up this process, selecting which folding, inserting and franking machine to use and configure each of these machines, setting the mail's size, weight, priority, and the carrier to use. These configurations can be set manually, using the machine's built-in displays and buttons. More convenient, however, is to configure the mailing process via one of the desktop applications Neopost provides.

The XBus. To connect a desktop application to the various machines, a software bus, called the XBus, has been developed. The XBus communicates over TCP and allows clients to discover other clients, announce provided services, query for

[1] Alternatively, a combined folding/inserting machine can be used.

services provided by other clients and subscribe to services. Also, XBus clients can send self-defined messages across the bus.

When this project started, an older version of the XBus existed, called the XBus version 1.0. Goal of our project was to re-implement the XBus while maintaining backward compatibility, i.e. the XBus 2.0 must support XBus 1.0 clients. Key requirements for the new XBus were improved maintainability and testability.

2.2 Model-Based Testing

Model-based testing Model-based testing (MBT, a.k.a. model-driven testing) is an innovative testing methodology that provides methods for automatic test generation, execution and evaluation. Model-based testing requires a formal model m, usually a transition system, of the system-under-test (SUT, a.k.a. implementation-under-test or IUT). This model m pins down the desired system behavior in an unambiguous way: traces of m are correct system behaviors, and traces not in m are incorrect.

The concept of model-based testing is visualized in Fig. 1. Tests derived from a model m are applied to the SUT, and based on observations made during test executions, a verdict (**pass** or **fail**) about the correctness of the SUT is given.

Each test case consists of a number of test steps. Each test step either applies a stimulus (i.e. an input to the SUT), or obtains an observation (i.e. a response from the SUT). In the latter case, we check whether the response was expected, that is, if it was predicted by the model m. In case of an unexpected observation, the test case ends with verdict **fail**. Otherwise, the test case may either continue with a next test step, or it may end with a verdict **pass**.

Test execution requires an adapter, which is a component of the tester in Fig. 1. Its role is to translate actions in the model m to concrete commands—in our case to TCP messages—of the SUT. Writing an adapter can be tricky, for instance if one action in the model corresponds to multiple actions in the system. Therefore, it was an important design rationale us to keep the adapter simple, which we achieved via a close correspondence between m and the system architecture.

However, given a model m and the adapter a, model-based testing is fully automatic. MBT tools can fully automatically derive test cases from the model, execute them, and issue verdicts. There are various MBT tools around, like SpecExplorer from Microsoft [34], Conformiq Qtronic [1], and AGEDIS [26]. Each of these tools varies in the capabilities, modeling languages and underlying theories, see [8,25] for an overview.

JTorX. These techniques have been implemented in the model-based test tool JTorX. JTorX [7,3] was (partly) developed during the Quasimodo project [6]. It improves over its predecessor TorX [9,33]—which was one of the first model-based testing tools in the field. JTorX is based on newer theory, and much easier to install, configure and use. Moreover, it has built-in adapter functionality to connect the model to the SUT via TCP/IP. All this turned out to be particularly helpful in this case study.

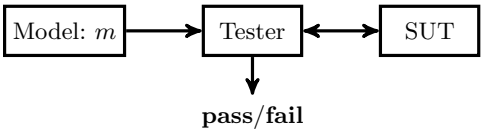


Fig. 1. Model-based testing

JTorX has built-in support for models in graphml [21], the Aldebaran (.aut) file format, and the Jararaca [2] file format. Moreover, it is able to access models via the mCRL2 [23], LTSmin [11] and CADP [20] tool environments.

In JTorX the test derivation and test execution functionalities are tightly coupled: test cases and test steps are derived on demand (only when required) during test execution. This is why we do not explicitly show test cases in Fig. 1.

Correctness of tests. MBT provides a rigorous underpinning of the test process: it can be shown that, under the assumption that the model correctly reflects the desired system behavior, all test cases derived from the model are correct, i.e., they yield the correct verdict when executed against any implementation, see e.g. [32]. More technically, the test case derivation methods underlying JTorX are provably correct, i.e. have been shown *sound* and *complete*. That is, any correct implementation of a model m will pass all tests derived from m (soundness). Moreover, for any incorrect implementation of m, there is at least one test case derivable from m that exhibits the error (completeness). Note that completeness is merely an important theoretical property, showing that the test case derivation method has no inherent blind spots. In practice only a finite number of test cases are executed. Hence, the test case exhibiting the error may or may not be among those test cases that are executed. As stated by the famous quote by Dijkstra: "testing can only show the presence of errors, not their absence".

Rich and well-developed MBT theories exist for control-dominated applications, and have been extended to test real-time properties [16,28,13], data-intensive systems [18], object-oriented systems [22], and systems with measure imprecisions [12]. Several of these extensions have been developed during the Quasimodo project as well.

2.3 The Specification Language mCRL2

The language mCRL2 [23,4] is a formal modeling language for describing concurrent systems, developed at the Eindhoven University of Technology. It is based on the process algebra ACP [10], and extends ACP with rich data types and higher-order functions. The mCRL2 toolset facilitates simulation, analysis and visualization of behavior; model-based testing against mCRL2 models is supported by the model-based test tool JTorX. Specifications in mCRL2 start with a definition of the required data types. Technically, the behavior of the system is declared via process equations of the form $X(x_1 : D_1, x_2 : D_2, x_n : D_n) = t$, where x_i is a variable of type D_i and t is a process term, see the example in Section 3.2. Process terms are built from potentially parameterized actions and

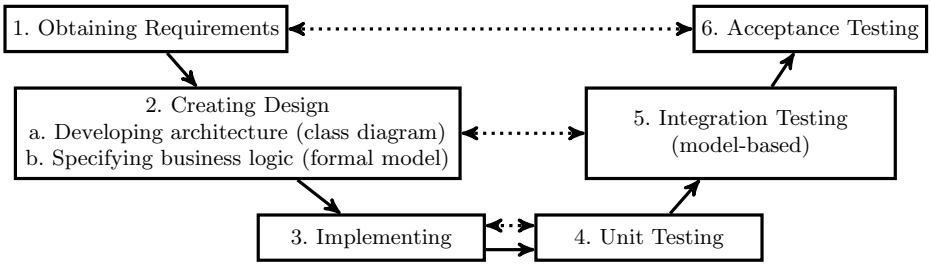


Fig. 2. The V-model that was used for development of XBus

the operators alternative composition, sum, sequential composition, conditional choice (if-then-else), parallel composition, and encapsulation, renaming, and abstraction. Actions represent basic events (like sending a message or printing a file) which are used for synchronization between parallel processes. Apart from analysis within the tool set, mCRL2 interoperates with other tools: Specifications in mCRL2 can be model checked via the CADP model checker, by generating the state space in `.aut` format, they can be proven correct using e.g. the theorem prover PVS, and they can be tested against with JTorX.

3 Development of the XBus

We developed the XBus implementation using the classical V-model [31], see Fig. 2. In our approach we have three testing phases: unit testing, integration testing and acceptance testing.

The sequel describes the activities carried out in each phase of the V-model. Each section below corresponds to an activity in the V-model. As stated, the total running time of the XBus development was 14 weeks.

3.1 XBus Requirements

We have obtained the functional and nonfunctional requirements by studying the documentation of the XBus version 1.0 (a four page English text document) and by interviewing the manager of the XBus development team.

The functional requirements express that the XBus is a centralized software application which can be regarded as a network router. Clients can connect and disconnect at any point in time. Connected clients can send XML-formatted messages to each other. Moreover, clients can discover other clients, announce services, and query for services that are provided by other clients. Also, they can subscribe to services, and send self-defined messages to each other. Below, we summarize the functional requirements; as said before, important non-functional requirements are testability, maintainability and backwards compatibility with the XBus 1.0.

Functional requirements are as follows.

1. XBus messages are formatted in XML, following the same Schema as the XBus 1.0.
2. Clients connecting to XBus perform a handshake with the XBus server. The handshake consists of a $\mathsf{Conn}_{\mathsf{req}}$—$\mathsf{Conn}_{\mathsf{ack}}$—$\mathsf{Conn}_{\mathsf{auth}}$ sequence.
3. Newly connected clients are assigned unique identifiers.
4. Clients can subscribe to be notified when a client connects or disconnects.
5. Clients can send messages to other clients with self-defined, custom, data. Such messages can have a self-defined, custom message type. In addition there are protocol messages for connecting, service subscription, service advertisement.
6. Clients can subscribe to receive all messages, sent by other clients, that are of one or more given types (including self-defined messages), using the Sub message.
7. Clients are able to announce services they provide, using the $\mathsf{Serv}_{\mathsf{ann}}$ message.
8. Clients can inquire about services, by specifying a list of service names in a $\mathsf{Serv}_{\mathsf{inq}}$ message. Service providers that provide a subset of the inquired services will respond to this client with the $\mathsf{Serv}_{\mathsf{rsp}}$ message.
9. Clients can send *private* messages, which are only delivered to a specified destination.
10. Clients can send *local* messages, which are delivered to the specified address, as well as to clients subscribed to the specified message type.

XBus protocol messages are the following.

$\mathsf{Conn}_{\mathsf{req}}$ (implicit) implied by a client establishing a TCP connection with XBus
$\mathsf{Conn}_{\mathsf{ack}}$ sent from XBus to a client just after the client establishes a TCP connection with the XBus, as part of the handshake.
$\mathsf{Conn}_{\mathsf{auth}}$ sent from a client to the XBus to complete the handshake.
(Un)Sub sent from a client to XBus, with as parameter a list of (custom) message types, to (un)subscribe receipt of all messages of the given types.
$\mathsf{Notif}_{\mathsf{conn}}$ sent from XBus to clients that subscribed connect notifications.
$\mathsf{Notif}_{\mathsf{disc}}$ sent from XBus to clients that subscribed disconnect notifications.
$\mathsf{Serv}_{\mathsf{ann}}$ sent (just after connecting) from a client c to XBus, which broadcasts it to all other connected clients, to announce the services provided by c.
$\mathsf{Serv}_{\mathsf{inq}}$ sent (just after connecting) from client to XBus, which broadcasts it to all other connected clients, to ask what services they provide.
$\mathsf{Serv}_{\mathsf{rsp}}$ sent from a client via XBus to another client, as response to $\mathsf{Serv}_{\mathsf{inq}}$, to tell the inquirer what services the responding client provides.

3.2 XBus Design

The design phase encompassed two activities: we created an architectural design, given by the UML class diagram in Fig. 3, and we made an mCRL2 model, describing the protocol behavior. An important feature of the UML design is that is already catered for model-based testing.

The architectural design and the mCRL2 model were developed in parallel; central in their design are the XBus messages: each message translates into a method in the class diagram and into an action in the mCRL2 model. The UML diagram specifies which methods are provided, and the mCRL2 model describes the order in which actions should occur, i.e. the order in which methods should be invoked. Thus, the architectural model in UML and the behavioral model in mCRL2 are tightly coupled.

Architectural Design. The architecture of the XBus is given in Fig. 3, following a standard client-server architecture. Thus, the XBus has a client side, implemented by the XBusGenericClient, and a server side, implemented by the XBusManager. The latter handles incoming protocol messages and sends the required responses. Both the server and the client use the communications package, which implements communication over TCP.

We have catered for model-based testing already in the design: the XBusManager has a subclass JTorXTestableXBusManager. As we elaborate in Section 3.5, the JTorXTestableXBusManager class overrides the send message from the XBusManager, allowing JTorX to have more control over the state of the XBus server.

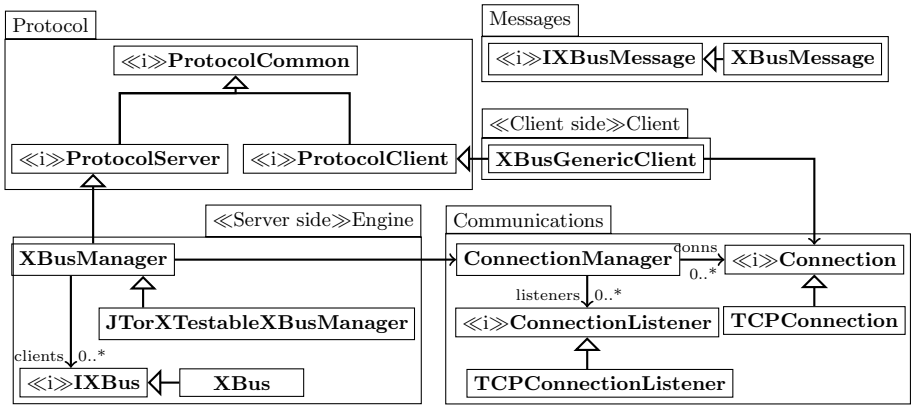


Fig. 3. High level architecture of the XBus system. It contains a server side package, and a client side package. Furthermore, it has functionality for TCP connections and XBus messages. Both server and client implement the Protocol abstract class. All interfaces are indicated with ≪i≫.

Modeling strategy. When creating the model, the first step is to define *what* and *what not* to model, to determine the abstraction level and boundaries of the model.

Included in the model. The messages that come into the server, their handling and their response messages are modeled. The handling of the messages is modeled as follows. After a message is received, the server will handle it. This means

that the server will send a response, relay the message, broadcast a message, and/or modify its internal state, depending on the type of message that arrived. Furthermore, the server keeps track of the client's state by keeping an internal list of client objects, just as in the Engine package in the architecture.

Excluded from the model. The Communications package is not included in the model. The model just describes that messages are received by and sent from the server (i.e. the XBusManager). This corresponds to the hand-over of incoming messages (from the perspective of XBus) from the Communications package to the Engine package, and the hand-over of outgoing messages in the opposite direction. So, the boundary between packages in the architecture corresponds with the boundary between the model and its environment.

Thus, we do *not* model internal components like TCP-sockets, queues, or programmatic events. The correctness of these internal components will be verified by unit tests. We will use the model discussed here to simulate and test the XBus protocol (i.e. the business logic).

The XBus model. We modeled the desired behavior of the XBus as an mCRL2 process. We chose mCRL2 because of its powerful data types, which make the modeling of the messages and its parameters convenient. In particular, we benefitted from its concise notation for enumerated types, records, and lists, and the ability to define functions. Functions are defined in a functional programming style. They can be called from the server process, and can modify data structures defined in the data part.

Data. All the data that the server keeps track of is kept in one data object: a list of clients. This is modeled as a list of data structures, that for each client contains the following items:

- an integer that represents the identity of the client;
- the connection status of the client, which is an enumeration of: disconnected, awaitingAuthentication, connected;
- the subscriptions of the client, which is a list of message types.
- the services that the client provides, which is a list of integers.

We defined functions to model manipulations on these data types. Most of our functions operate on the lists of clients and the client's lists of subscriptions and services. Typical operations are insertion, lookup, update, and removal of items in these lists.

Behavior. The behavior of the XBus server is modeled as a single process that—for all kinds of incoming messages that it may receive—accepts a message, processes it (which may involve an update of its state), and sends a response (where appropriate), after which it is ready to accept the next message.

Listing 1 shows part of the definition of this process (slightly simplified). The process is named listening. It has a single parameter: c, the list of clients. The fragment shows that for each client—where j is used as index in c (line 2)— that

```
1 proc listening(c:Clients) =
2    (sum j:Int.(j >= 0 && j < numClients(c) &&
3                getClientStatus(j, c) == DISCONNECTED )
4       -> (ConnectRequest.ConnectAcknowledge.
5           listening(changeClientStatus(j, c, AWAIT_AUTH)))
6       <> delta
7    ) + ...
```

Listing 1. Definition of XBus handling of $\mathsf{Conn}_{\mathsf{req}}$ message in mCRL2

currently is in `DISCONNECTED` state (line 3), the server is willing to accept a ConnectRequest message, after which it will send out a ConnectAcknowledge message (line 4), after which it updates the status of the $\mathtt{j}^{th}$ client in the list to `AWAIT_AUTH` and continues processing—modeled by the recursive call to listening with the updated client list (line 5).

The language mCRL2 allows modeling of systems with multiple parallel processes, but this is not needed here. Having multiple concurrent processes would make the system as well as the model more complicated, which would make them harder to maintain and test. One might choose to use multiple processes when performance of the system is expected to be a problem, but that is not an issue here. In a large mailing room there may be 20 clients at the same time, a number with which the single-process server can easily cope.

Model size. The entire model consists of 6 pages of mCRL2, including comments. Approximately half of it concerns the specification of data types and functions over them; the other half is the behavioral specification.

Model validation. During the construction of the model, we exhaustively used the simulator from the mCRL2 toolkit. We incrementally simulated smaller and larger models, where we used both manual and random simulation. This was done for two reasons. First, to get a better understanding of the working of the whole system, and to validate the design already before the implementation phase was started. This was particularly useful to improve our understanding of the XBus protocol, of which only a (non-formal) English text description was available, which contained several ambiguities. Second, to validate the model, to be sure that it faithfully represents the design, i.e. to fulfill the assumptions stated in Section 2.2, such that when we use JTorX to test our implementation against the model, all tests that JTorX derives from the model will yield the correct verdict. Due to time constraints, model-checking was not performed. It would have allowed validation of (basic) properties like the absence of deadlocks, as well as checking more advanced properties, e.g. that every message sent eventually reaches the specified destination(s).

3.3 Implementation

Once we had sufficient confidence in the quality of the design—to a large extent due to modeling and simulation—it was implemented. The programming language used was C#—use of .NET is company policy.

3.4 Unit Testing

As mentioned in the introduction, the overall test strategy was to test data behaviour using unit testing, and protocol behaviour using model-based testing. The classes in the Communications and Messages packages were therefore tested using unit testing.

For the Communications package unit tests were written to test the ability to start a TCP listener and to connect to a TCP listener, to test the administration of connections, and to test transfer of data. For the Messages package unit tests were written to test construction, parsing and validation of messages. The latter was tested using both correct and incorrect messages.

Each error that was found during unit testing was immediately repaired.

3.5 Model-Based Integration Testing

After unit testing of data behaviour, we used model-based testing for the business logic, i.e. to test the interaction between XBus and its clients. This is because here the dynamic behavior, i.e., the order of protocol messages, crucially determines the correctness of the protocol.

Test architecture. To test whether the XBus interacts correctly with its environment, we chose a test set up with 3 XBus clients, see Fig. 4. Thus, JTorX plays the role of 3 XBus clients, which are able to perform all protocol actions described in Section 3.1. We first discuss how we connected JTorX to the XBus server, and then we briefly discuss an alternative.

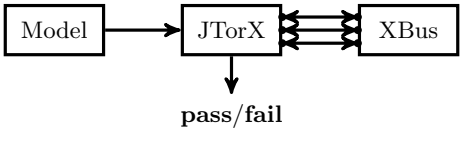


Fig. 4. Testing XBus with JTorX playing the role of 3 clients

Our solution is depicted in Fig. 5. We provide stimuli to the XBus using three XBusGenericClient instances, each of which is connected to the XBus via TCP. We observe the responses from the XBus not via the XBusGenericClient, but via a direct (testing) interface that has been added to XBus. This interface is provided by the JTorXTestableXBusManager in the Engine package, see Fig. 3. JTorXTestableXBusManager overrides the function that XBus uses to send a message to a specified client, and instead logs the message name and relevant parameters in the textual format that JTorX expects. Additional glue code—the adapter—provides the connection between JTorX and the XBusGenericClient instances on the one hand, and between JTorX and XBus test interface on the other hand. From JTorX the adapter receives requests to apply stimuli, and from the XBus test interface it receives observed responses. The adapter forwards the received responses to JTorX without additional processing. For each

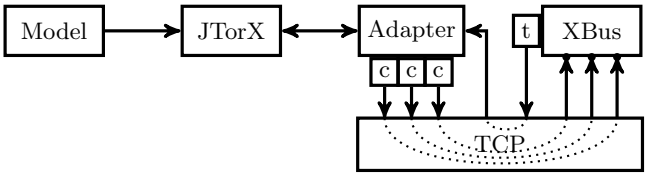


Fig. 5. The Test Architecture that we used: JTorX provides stimuli to XBus via generic clients (c) over TCP, and observes responses via test interface (t), also connected via TCP

received request to apply a stimulus the adapter uses XBusGenericClient methods to construct a corresponding XBusMessage message and send it to the XBus server (except for the $\mathsf{Conn}_{\mathsf{req}}$ message, for which XBusGenericClient only has to open a connection to XBus).

The adapter is implemented as a C# program that uses the Client package (see Fig. 3) to create the three XBusGenericClient instances, which in turn use the Communications package to interact with the XBus. The main functionality that had to be implemented was the mapping between XBus messages and the corresponding XBusGenericClient methods, and the corresponding XBusGenericClient instances. Due to the one-to-one mapping that exists between these—by design, recall Section 3.2— implementing this mapping was rather straightforward.

JTorX and the adapter communicate via TCP: the adapter works as a simple TCP server to which JTorX connects as a TCP client. This is one of two possibilities offered by JTorX; which of the two is chosen does not really matter.

It may seem that the Communications package does not play a role during model-based testing with this test architecture, also because we mentioned that we excluded it from the model. However, the Communications package is used normally in the XBus to receive the messages that clients send to it. Moreover, the only functionality of the Communications package that is not used in the XBus itself in this test architecture—the functionality to send messages over TCP—is used by the XBusGenericClient instances that are used to send the stimuli to the XBus.

An alternative approach would have been to not add the direct (testing) interface to XBus, to observe its responses, but to use the XBusGenericClient instances for this, as depicted in Fig. 6. This alternative approach has the clear advantage that no additional (testing) interface has to be added to XBus, and thus the interaction via the XBusGenericClient instances is (in the perception of XBus) identical to the interaction during deployment.

However, this alternative approach also has one slight disadvantage. From the perspective of an observer of XBus responses, each of the TCP connections between an XBusGenericClient instance and the XBus resembles a first-in first-out (FIFO) queue, where a message that is sent later, over one connection, may overtake a message that was sent earlier over another connection. This means that the order in which XBus responses, via XBusGenericClient instances, arrive

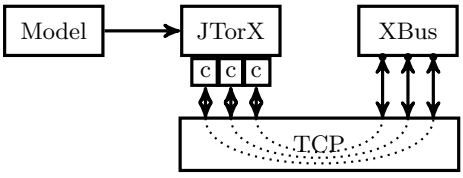


Fig. 6. Alternative solution: JTorX connected to XBus via generic clients (c) over TCP

at the adapter, and thus ultimately at JTorX, may differ from the order in which XBus sends them. This, in turn, may result in incorrect fail verdicts—because the model does not reflect the FIFO-queue behavior of the TCP communication medium between XBus and the adapter. We have seen this reordering effect before, for example in our experiments with model-based testing of a simple chatbox protocol entity [9] and know that we can deal with it by extending the model with FIFO buffers that model the FIFO queue behavior of the TCP connections.

With both test architectures we have to deal with the reordering effect of the TCP connections, either by extending the XBus with a specific testing interface, or by extending the model. In this case we chose to extend the XBus.

Running JTorX. Once we had the model, the XBus implementation to test, and the means to connect JTorX to it, testing was started. We ran JTorX in random mode. Figure 7 shows the settings in JTorX. These include the location of the model file, the way in which the adapter and the XBus are accessed, and an indication of which messages are input (from the XBus server perspective) and which ones are output.

Bugs found using JTorX. One of the most interesting parts of testing is finding bugs. In this case, not only because it allows improving the software, but also because finding bugs can be seen as an indication that model based testing is actually helping us. We found 5 bugs. Typically these were found within 5 minutes after the start of a test. Some of them are quite subtle:

1. The $\mathsf{Notif}_{\mathsf{disc}}$ message was sent to unsubscribed clients. This was due to an if-statement that had a wrong branching expression.
2. The $\mathsf{Serv}_{\mathsf{ann}}$ message was sent (also) to unauthorized clients. Clients that were still in the handshake process with the server, and thus not fully authenticated, received the $\mathsf{Serv}_{\mathsf{ann}}$ message. To trigger this bug one client has to (connect and) announce its service while another client is still connecting.
3. The message subscription administration did not behave correctly: a client could subscribe to one item, but *not* to two or more. This was due to a bug in the operation that added the subscription to the list of a client.
4. The same bug also occurred with the list of provided services. It was implemented in the same way as the message subscription administration.

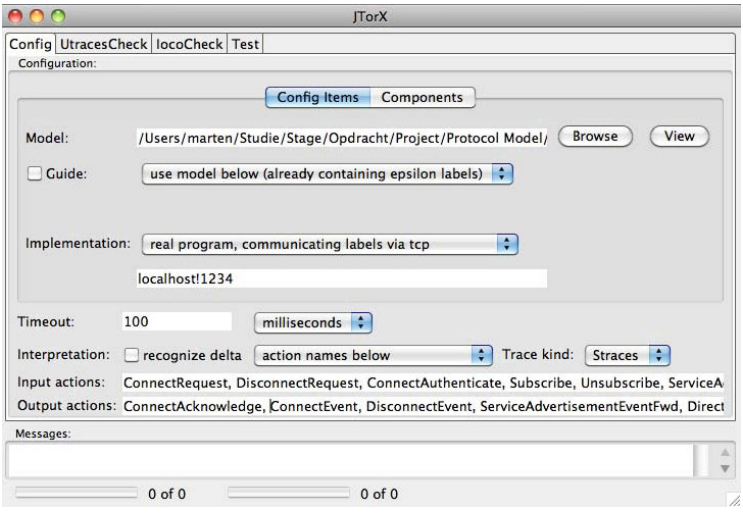


Fig. 7. Screen shot of the configuration pane of JTorX, set up to test XBus. JTorX will connect to (the adapter that provides access to) the system under test via TCP on the local machine, at port 1234. The bottom two input fields list the input and output messages.

5. There was a flaw in the method that handles Unsub messages. The code that extracts subscriptions from these messages (to be able to remove them from the list of subscriptions of the corresponding client) contained a typing error: two terms in an expression were interchanged.

All these bugs concern the order in which protocol messages must occur. Therefore, it is our firm belief that they are much harder to discover with unit testing.

3.6 Acceptance Testing

Acceptance testing was done in the usual way. We organized a session with the manager of Neopost's ISS group, and showed how the XBus 2.0 worked. In particular, we demonstrated that it implements the features required in Section 3.1.

4 Findings and Lessons Learned

In a time perspective. So how long did it take to create the artefacts for model-based testing, namely the model, the test interface and the adapter? Programming and simulating the model took 2 weeks, or 80 hours. The test interface was created in a few hours, since it was designed to be loosely coupled to the engine. It was a matter of a few dozens lines of code. The adapter was created in two days, or 16 hours. Thus, given the total project time of 14 weeks, creating the artefacts needed for model-based testing took thus about 17% of our time.

The modeling process. Writing a model takes a significant amount of time, but also forces the developer to think about the system behavior thoroughly. Moreover, we found it extremely helpful to use simulation to step through the protocol, before implementing anything. Making and simulating a model gives a deep understanding of the system, in an early stage of development, from which the architectural design profits.

Automated testing with JTorX. Writing an adapter can be a large project, but in this case it was relatively straightforward. This can be attributed to having an architectural design that closely resembles the formal model, and a one-to-one mapping between the actual XBus messages and their model representation.

5 Conclusions and Future Research

We conclude that model-based testing using JTorX was a success: with a relatively limited effort, we found five subtle bugs. We needed 17% of the time to develop the artefacts needed for model-based testing, and given the errors found, we consider that time well spent. Moreover, for future versions of the XBus, JTorX can be used for automatic regression tests: by adapting the mCRL2 model to new functionality, one can detect automatically if new bugs are introduced.

We also conclude that making the formal model together with the architectural design had a positive effect on the quality of the design. Moreover, the resulting close resemblance between model and design simplified the construction of the adapter.

Although construction of the adapter was relatively straightforward, it would have been even easier if (parts of) the adapter could have been generated automatically, which is an important topic for future research. Another very important topic is to implement test coverage metrics (e.g. from [14]) in JTorX, so that we can quantify how thoroughly we have been testing.

References

1. Conformiq webpage (April 2011), `http://www.conformiq.com/`
2. Jararaca manual, `http://fmt.cs.utwente.nl/tools/torx/jararaca.1.html`
3. JTorX webpage (August 2009), `http://fmt.ewi.utwente.nl/tools/jtorx/`
4. mCRL2 toolkit webpage (September 2009), `http://www.mcrl2.org/`
5. Neopost Inc. webpage (August 2009), `http://www.neopost.com/`
6. Quasimodo webpage (August 2011), `http://www.quasimodo.aau.dk/`
7. Belinfante, A.: JTorX: A tool for on-line model-driven test derivation and execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
8. Belinfante, A., Frantzen, L., Schallhart, C.: Tools for test case generation. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 391–438. Springer, Heidelberg (2005)

9. Belinfante, A., et al.: Formal test automation: A simple experiment. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) IWTCS 1999, pp. 179–196. Kluwer, Dordrecht (1999)
10. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes. In: de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds.) Proceedings of the CWI Symposium on Mathematics and Computer Science. CWI, Amsterdam (1985)
11. Blom, S.C.C., van de Pol, J.C., Weber, M.: Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology, University of Twente, Enschede (2009)
12. Bohnenkamp, H.C., Stoelinga, M.I.A.: Quantitative testing. In: Proceedings of the 7th ACM International Conference on Embedded Software, pp. 227–236. ACM, New York (2008)
13. Brandán Briones, L.: Theories for Model-based Testing: Real-time and Coverage. Ph.D. thesis, University of Twente (March 2007)
14. Brinksma, E., Briones, L.B., Stoelinga, M.I.A.: A semantic framework for test coverage. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 399–414. Springer, Heidelberg (2006)
15. Cofer, D., Fantechi, A. (eds.): FMICS 2008. LNCS, vol. 5596. Springer, Heidelberg (2009)
16. David, A., Larsen, K.G., Li, S., Nielsen, B.: Timed testing under partial observability. In: ICST, pp. 61–70. IEEE Computer Society, Los Alamitos (2009)
17. Ferrari, A., Grasso, D., Magnani, G., Fantechi, A., Tempestini, M.: The metrô rio atp case study. In: Kowalewski, S., Roveri, M. (eds.) [27], pp. 1–16
18. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A symbolic framework for model-based testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
19. Garavel, H., Viho, C., Zendri, M.: System design of a cc-numa multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. STTT 3(3), 314–331 (2001)
20. Garavel, H., et al.: Cadp 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
21. GraphML work group: GraphML file format, `http://graphml.graphdrawing.org/`
22. Grieskamp, W., Qu, X., Wei, X., Kicillof, N., Cohen, M.B.: Interaction coverage meets path coverage by SMT constraint solving. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM 2009. LNCS, vol. 5826, pp. 97–112. Springer, Heidelberg (2009)
23. Groote, J.F., et al.: The mCRL2 toolset. In: Proc. International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008 (2008)
24. Hansen, H.H., Ketema, J., Luttik, S.P., Mousavi, M.R., van de Pol, J.C.: Towards model checking executable uml specifications in mcrl2. Innovations in Systems and Software Engineering 6(1-2), 83–90 (2010)
25. Hartman, A.: Model based test generation tools survey. Tech. rep., Centre for Telematics and Information Technology, University of Twente (2009)
26. Hartman, A., Nagin, K.: The agedis tools for model based testing. SIGSOFT Softw. Eng. Notes 29, 129–132 (2004)
27. Kowalewski, S., Roveri, M. (eds.): FMICS 2010. LNCS, vol. 6371. Springer, Heidelberg (2010)
28. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using uppaal: Status and future work. In: Perspectives of Model-Based Testing. Dagstuhl Seminar Proceedings, vol. 04371 (2004)

29. Leveson, N.G.: Experiences in designing and using formal specification languages for embedded control software. In: Lynch, N.A., Krogh, B.H. (eds.) [30], p. 3
30. Lynch, N.A., Krogh, B.H. (eds.): HSCC 2000. LNCS, vol. 1790. Springer, Heidelberg (2000)
31. Rook, P.E.: Controlling software projects. IEE Software Engineering Journal 1(1), 7–16 (1986)
32. Timmer, M., Brinksma, E., Stoelinga, M.: Model-based testing. In: Software and Systems Safety: Specification and Verification. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, Amsterdam (2011)
33. Tretmans, J., Brinksma, H.: TorX: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp. 31–43 (December 2003)
34. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008)

Symbolic Power Analysis of Cell Libraries

Matthias Raffelsieper and MohammadReza Mousavi

Department of Computer Science, TU/Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{M.Raffelsieper,M.R.Mousavi}@tue.nl

Abstract. Cell libraries are collections of logic cores (cells) used to construct larger chip designs; hence, any reduction in their power consumption may have a major impact in the power consumption of larger designs. The power consumption of a cell is often determined by triggering it with all possible input values in all possible orders at each state. In this paper, we first present a technique to measure the power consumption of a cell more efficiently by reducing the number of input orders that have to be checked. This is based on symbolic techniques and analyzes the number of (weighted) wire chargings taking place. Additionally, we present a technique that computes for a cell all orders that lead to the same state, but differ in their power consumption. Such an analysis is used to select the orders that minimize the required power, without affecting functionality, by inserting sufficient delays. Both techniques have been evaluated on an industrial cell library and were able to efficiently reduce the number of orders needed for power characterization and to efficiently compute orders that consume less power for a given state and input-vector transition.

1 Introduction

A *cell library* is a collection of logic cores used to construct larger chip designs, consisting of combinational cells (e.g., `and` and `xor`) and sequential cells (e.g., latches and flip-flops). Cell libraries are usually described at multiple levels of abstraction, such as a transistor netlist and a Verilog description. Cells are used repeatedly in many larger designs and hence any minor improvement in their power consumption may lead to considerable power saving in the subsequent designs.

In this paper, we analyze the dynamic power consumption of netlists, as these describe the design that will finally be manufactured. An analysis at the level of Verilog descriptions is also possible but not very promising, because the same functional behavior (as that of the netlist) is often described using very different structures, a prominent example being User Defined Primitives (UDPs). To measure the dynamic power consumed by a netlist we use an abstract measure, namely the number of wires charged, possibly weighted by the node capacitance, if this is additionally available.

From a given transistor netlist, we first build a transition system, which describes the state of each wire in the netlist. In practice, this transition system is described symbolically by equations in the inputs and values of wires [2]; hence,

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 134–148, 2011.

we use Binary Decision Diagrams (BDDs) for a symbolic representation of the netlist semantics. In order to efficiently analyze the different power consumption of different permutations, we quotient these permutations into equivalence classes. Every such equivalence class contains different orders of applying a transition from one input vector to another, but all of these orders have the same functional effect and consume the same amount of power. Thus, during power characterization, only one of these orders has to be considered.

In practice, the order of evaluating input changes may be controlled efficiently; in such cases, we seek a reduction in the dynamic power consumption by choosing, among functionally equivalent orders, the one that has the minimal power consumption. For this problem, we developed an analysis of functionally equivalent orders. Then, given the current state of the wires and an input vector transition, we can determine the order that consumes the minimal amount of power. Thus, by choosing this order the functionality of the circuit is not altered, but the power consumption is indeed reduced.

Related Work. The work reported in [3] also determines the power consumption of cells. The authors present an empirical algorithm, which also groups together different input vector transitions. However, they group together different values of inputs, whereas our approach groups together different orders of applying the same input vector. Furthermore, their grouping is made manually and afterwards all remaining input vectors and orders are enumerated explicitly, as opposed to our symbolic approach. Another approach that also uses a transition system model of circuits is presented in [8]. This approach builds an explicit representation of the transition system, and hence has to combat the size of these transition systems by simplifying the netlist, something which is not required in our symbolic representation. A symbolic representation of cells for the purpose of power analysis is also used in [1]. There, the symbolic representation is used during simulation of cells to determine the charge for each wire. Our work can be seen as a preprocessing step to theirs, as we first reduce the number of orders that later have to be simulated. Already in [7] it was observed that superfluous transitions (called *glitches*) of signals cause an increased power consumption. In contrast to our work, there glitches are detected by simulations, and only considered at cell outputs. The authors propose a number of techniques to reduce glitches. One of these is the addition of delays to enforce a certain order of events, which is also what we propose to select a low power evaluation. The theoretical basis of this paper builds upon [5] and [6]. Those analyses are extended by also taking power consumption into account and using the results to build symbolic graph structures that represent the equivalence classes of orders in a compact way.

Paper Structure. The rest of this paper is structured as follows. In Section 2, we introduce the notion of vector-based transition system which we use to model the semantics of transistor netlists. Furthermore, the section introduces orders as permutations of inputs and lists as their building blocks. Section 3 then presents our first technique to determine all equivalence classes of orders that are functionally equivalent and consume the same amount of power. A technique that determines

equivalence of orders based on functional equivalence is then presented in Section 4. For each of these equivalence classes, it is furthermore analyzed which of these orders consumes the least amount of power for a given state and input vector transition. We briefly sketch our implementation in Section 5 and report empirical results obtained from applying our implementation on an industrial cell library in Section 6. We conclude the paper in Section 7.

2 Preliminaries

The analysis in this paper is concerned with *transistor netlists*. These consist of a number of transistor instantiations, which for the purpose of this paper are seen as a switch. From such a transistor netlist, a system of Boolean equations is created, using the method of [2]. These equations form a transition system, where states are represented by a vector of Boolean variables. Transitions occur between stable states (i.e., states that are finished evaluating for the current input values) and are labeled with another Boolean vector, representing the new values of the inputs. The structure of the states is irrelevant for the transitions, thus the set of states is kept abstract in our formal treatment. Only in the experiments, state vectors are investigated to determine the number of charged wires. Since a transistor netlist can start up in any arbitrary state, the transition systems we consider do not have an initial state, instead an evaluation can start in any state.

Hence, we consider a *vector-based transition system*, which is a triple $T = (S, I, \rightarrow)$ where S is an arbitrary set of states (usually, as explained above, S is a set of vectors representing the current internal state), $I = U^m$ is the set of *input vectors* for some basic set U (commonly the Boolean values $\mathbb{B}$), and $\rightarrow \subseteq S \times I \times S$ is the transition relation. For a vector $\vec{v} = (v_1, \ldots, v_k)$, we denote its j-th position by $\vec{v}|_j = v_j$ and the update of the vector $\vec{v}$ in coordinate j by value v' is denoted by $\vec{v}[j := v'] = (v_1, \ldots, v_{j-1}, v', v_{j+1}, \ldots, v_k)$.

We are only interested in one-input restricted traces, because they form the common semantic model of hardware description languages (the so-called *single event* assumption in [3]). In order to define *one-input restricted traces*, i.e., traces where the used input vectors differ in at most one coordinate, we use the *Hamming distance* $\mathrm{d_H}(\vec{i}_1, \vec{i}_2) = |\{1 \leq j \leq m : \vec{i}_1|_j \neq \vec{i}_2|_j\}|$. A one-input restricted trace is then a trace consisting of consecutive steps $s_0 \xrightarrow{\vec{i}_1} s_1 \xrightarrow{\vec{i}_2} s_2$ with $\mathrm{d_H}(\vec{i}_1, \vec{i}_2) \leq 1$. We define the transition system $T^I = (S \times I, \rightarrow_{T^I})$ as the system where the states are extended with the input vector used to arrive in that state, and the transitions are only labeled by the input position that was changed. Formally, we have $\rightarrow_{T^I} \subseteq (S \times I) \times \{1, \ldots, m\} \times (S \times I)$ where $s_0;\vec{i}_0 \xrightarrow{j}_{T^I} s_1;\vec{i}_1$ iff $s_0 \xrightarrow{\vec{i}_1} s_1$ and $\vec{i}_1 = \vec{i}_0[j := \vec{i}_1|_j]$. Here and in the following we denote a tuple $(s, \vec{i}) \in S \times I$ by $s;\vec{i}$. It is easy to see that one-input restricted traces can be converted between the two representations. Thus, we use the two representations interchangeably in the rest of this paper and drop the subscript T^I if no confusion can arise.

For an evaluation of a netlist, we want to consider each input exactly once, to determine whether it has changed its value or not. Thus, we are interested in one-input restricted traces of length m where none of the indices occurs twice. This can be described by means of a *permutation*, that assigns to each step the coordinate of the input to consider. We write Π_m for the set of all permutations over the numbers $\{1, \ldots, m\}$. In the remainder, we will make use of the fact that every permutation can be obtained by a sequence of *adjacent transpositions*, which are swappings of two neighboring elements. To be able to construct permutations from smaller parts, we denote by $\mathcal{L}_m$ the set of all *lists* containing each of the numbers $\{1, \ldots, m\}$ at most once. Then, we interpret the permutations Π_m as those lists containing every number exactly once (i.e., the lists of length m). Given a list $\ell = j_1 : \cdots : j_k \in \mathcal{L}_m$, we define the *length* of this list as $|\ell| = k$ and the *sublist* $\ell[a \,..\, b] = j_a : \cdots : j_b$ for $1 \leq a \leq b \leq k$. The empty list is denoted $\emptyset$ and a singleton list is denoted by its only element. The concatenation of two lists $\ell_1 = j_1 : \cdots : j_a$ and $\ell_2 = j'_1 : \cdots : j'_b$ with $j_c \neq j'_d$ for all $1 \leq c \leq a$ and $1 \leq d \leq b$ is defined as $\ell_1 {+\!+} \ell_2 = j_1 : \cdots : j_a : j'_1 : \cdots : j'_b$.

3 Reducing Input Vector Orders for Power Analysis

3.1 Order-Independence of Power-Extended Transition Systems

Our first improvement in power analysis is achieved by grouping, in equivalence classes, those orders that have the same power characteristics. For such orders, it is sufficient to only consider one member of the equivalence class. Important for such equivalence classes is to result in the same state in order not to affect the functionality of the netlist. This latter problem of determining whether the state reached after applying an input vector in different orders is the same has already been considered in [5] and [6]. Here, we extend those works to also consider the dynamic power consumption. For this purpose, we define a *power-extended vector-based transition system.*

Definition 1. *The* power-extended vector-based transition system $T_p = (S_p, I, \rightarrow_p)$ *of a vector-based transition system* $T = (S, I, \rightarrow)$ *is defined* $S_p = \mathbb{R} \times S$ *and* $w;s \xrightarrow{\vec{i}}_p w{+}p(s,s');s'$ *for* $s \xrightarrow{\vec{i}} s'$. *The function* $p : S \times S \to \mathbb{R}$ *computes a weighted number of wire chargings given the source and target states.*

The added first component of the states, a number, is used to sum the weights of the charged wires (which we interpret as the consumed power) during an evaluation. (Our definition indeed allows for weighted wire charging in order to cater for node capacitance in our calculations. However, for presentation purposes throughout the rest of this paper, we assume the weights of all wire chargings to be equal; hence, in the remainder of the paper, the sum of weights denotes the number of wire chargings.) Note that a vector-based transition system does not assume a particular type of the states, so this is still a vector-based transition system.

We initially set the power component (the real number component in the state) to 0, indicating that no chargings have taken place yet. Then, we select

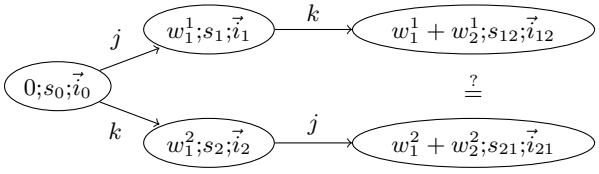


Fig. 1. Evaluation of two input coordinates

two inputs (identified by their position in the input vector, as in the transition system T^I) and change them in both possible orders. Finally we check whether the resulting states for the two orders are equal or not. For example, assume that we initially arrive in state s_0 with an input vector $\vec{i}_0$, denoted by $0;s_0;\vec{i}_0$. Then, we apply the two consecutive input changes, denoted by $[j := v'_j]$ and $[k := v'_k]$ where $j \neq k$, in the two possible orders leading to the two evaluations depicted in Figure 1.

As indicated in Figure 1, it remains to be checked whether the two states $w_1^1 + w_2^1;s_{12};\vec{i}_{12}$ and $w_1^2 + w_2^2;s_{21};\vec{i}_{21}$ are equal or not. First, we note that the input vectors $\vec{i}_{12}$ and $\vec{i}_{21}$ are equal, as they are constructed by updating positions j and k in $\vec{i}_0$ with the same values. Formally, this holds because for $j \neq k$, $\vec{i}_{12} = \vec{i}_0[j := v'_j][k := v'_k] = \vec{i}_0[k := v'_k][j := v'_j] = \vec{i}_{21}$. Thus, we only have to compare the remaining parts of the states. We have already addressed this problem, called *order-independence*, for generic vector-based transition systems in [6]. Checking that the states s_{12} and s_{21} are equal is the same as order-independence, i.e., checking that the order of these two inputs does not affect the functionality. By requiring that $w_1^1 + w_2^1 = w_1^2 + w_2^2$, we additionally require that the order of the two inputs also does not cause different power consumptions.

In this paper, by exploiting the result of [6], we show that for transistor netlists checking order-independence for two inputs is both necessary and sufficient to establish order-independence for traces of full input length. In order to do this we introduce the relation $\twoheadrightarrow$, which denotes traces.

Definition 2. *Let $T = (S, I, \rightarrow)$ be a vector-based transition system with m inputs, let $s;\vec{i} \in S \times I$, and let $\ell = j_1 : \cdots : j_k \in \mathcal{L}_m$.*

We define $\twoheadrightarrow_T \subseteq (S \times I) \times \mathcal{L}_m \times (S \times I)$ as $s;\vec{i} \stackrel{\ell}{\twoheadrightarrow}_T s';\vec{i}'$ iff there exists a state $s_0;\vec{i}_0$ and $1 \leq b \leq m$ such that $s_0;\vec{i}_0 \stackrel{b}{\rightarrow} s;\vec{i} \stackrel{j_1}{\longrightarrow} \ldots \stackrel{j_k}{\longrightarrow} s';\vec{i}'$.

In the above definition, we additionally require that the initial state is reachable from some other state, which we call *one-step reachability*. This restriction is added to rule out transient initial states that can only occur at boot-up and will never be reached again.

We then call a vector-based transition system T with m inputs *order-independent*, iff for all $\pi, \pi' \in \Pi_m$ it holds that $\stackrel{\pi}{\twoheadrightarrow} = \stackrel{\pi'}{\twoheadrightarrow}$. To relate order-independence, which considers traces of length m, and the check that only considers two inputs, we make use of the following theorem of [6].

Theorem 3 (Theorem 9 in [6]). *Let $T = (S, I, \rightarrow)$ be a deadlock-free vector-based transition system with m inputs having the fixed-point property. Then T is order-independent, iff $j \overset{\downarrow}{\diamond} k$ for all $1 \leq j < k \leq m$.*

In Theorem 3, deadlock-freedom has its intuitive meaning, namely that for every current state and every possible input transition, a next state can be computed. This is the case for the semantics of transistor netlists, our target application, as they always compute a next state for any input vector. The second requirement, the fixed-point property, demands that a reached state is stable, i.e., applying the same input vector twice does not result in a different state from when the input vector is only applied once. It is shown in [6] that vector-based transition systems constructed from a transistor netlist using the algorithm of [2] always have the fixed-point property. Intuitively, this holds because the states are stable and hence cannot distinguish the stable situation from applying the same inputs again. Since an unchanged state means that also the number of wire chargings does not change, this also holds in our power-extended vector-based transition system. Therefore, without mentioning this explicitly in the remainder of this paper, we assume these two hypotheses of Theorem 3 to hold.

The relation $\overset{\downarrow}{\diamond}$, called the *one-step reachable commuting diamond*, relates two input positions $1 \leq j \neq k \leq m$, if for every one-step reachable state $s_0;\vec{i}_0 \in S \times I$, $s_0;\vec{i}_0 \xrightarrow{j} \circ \xrightarrow{k} s_{12};\vec{i}_{12}$ iff $s_0;\vec{i}_0 \xrightarrow{k} \circ \xrightarrow{j} s_{12};\vec{i}_{12}$. This is similar to the situation depicted in Figure 1, assuming that the states s_{12} and s_{21} and their power consumptions are equal.

To summarize, also for power-extended vector-based transition systems we only have to analyze pairs of inputs, instead of complete sequences, to determine order-independence and therefore equivalent power consumption. This drastically decreases the number of required checks from $m!$, i.e., the number of all permutations, to $\frac{m^2-m}{2}$, the number of all pairs of inputs.

3.2 Equivalence Relation on Orders

Full order-independence of a power-extended vector-based transition system would mean that all orders always have the same power consumption. Of course, this is neither expected in any useful transistor netlist, nor is it of much practical relevance. Therefore, we want to create an equivalence relation on orders that groups together those subsets of orders having the same number of wire chargings. This relation, formally defined below, is called *power independence*.

Definition 4. *Let $T = (S, I, \rightarrow)$ be a vector-based transition system with m inputs.*

We define a relation $\leftrightarrow_T$ on $\mathcal{L}_m$, where $\ell \leftrightarrow_T \ell'$ iff the lists are equal except for swapped positions $\ell'[j+1] = \ell[j]$ and $\ell'[j] = \ell[j+1]$, for which the one-step reachable commuting diamond property holds (i.e., $\ell[j] \overset{\downarrow}{\diamond} \ell[j+1]$).

Using relation $\leftrightarrow_T$, we define the equivalence relation $\equiv_T$ on $\mathcal{L}_m$ as the reflexive transitive closure of $\leftrightarrow_T$. If $\ell \equiv_T \ell'$, then we also call ℓ and ℓ' (power-) independent.

Note that the above definition is using a general vector-based transition system. If this is a power-extended one, then we call the relation power-independence, otherwise we only talk about independence of lists. For the power-independence relation, we have the following result, showing that it indeed groups together those orders that have equal (functional and power consumption) behavior.

Lemma 5. *Let $T_p = (S_p, I, \rightarrow_p)$ be a power-extended vector-based transition system with m inputs and let $\pi, \pi' \in \Pi_m$ be power-independent (i.e., $\pi \equiv_{T_p} \pi'$).*

Then for traces $0;s_0;\vec{i}_0 \xrightarrow{\pi}\!\!\!\!\rightarrow_{T_p} w_1;s_1;\vec{i}$ and $0;s_0;\vec{i}_0 \xrightarrow{\pi'}\!\!\!\!\rightarrow_{T_p} w_2;s_2;\vec{i}$ it holds that $w_1 = w_2$ and $s_1 = s_2$.

Proof. Follows by an induction on the number of swapped input coordinates to reach π' from π. □

Thus, to characterize a cell, one only has to choose one representative from each power-independent equivalence class and measure the power consumption for this order. All other orders in this equivalence class will have the same power consumption and hence do not have to be considered.

To obtain the different orders that have to be considered, we build the *power-independence DAG* (directed acyclic graph) that enumerates all equivalence classes of the power-independence relation $\equiv_{T_p}$.

Definition 6. *Let $T_p = (S_p, I, \rightarrow_p)$ be a power-extended vector-based transition system with m inputs.*

The power-independence DAG *$G_i = (V_i, \rightarrowtail_i)$ of T_p is defined as $V_i \subseteq \mathcal{L}_m$ with root $\emptyset \in V_i$ and for $1 \leq j \leq m$ with $j \notin \ell$, $\ell \rightarrowtail_i \ell'$ for some unique $\ell' \equiv_{T_p} \ell{+\!+}j$.*

We did not label the edges with inputs in the above-given DAG since the input corresponding to each edge is always the single element by which the two lists of the start and the end node of the edge differ.

To construct the power-independence DAG G_i, we start with the single root of this DAG, $\emptyset$, which indicates that initially no inputs have been considered yet. The construction of the DAG then proceeds in a breadth-first fashion: for each *leaf* ℓ (which is a node without outgoing edges) and every input j that has not yet been considered (i.e., is not in ℓ), an edge is added to that leaf. The target node of this edge is determined by looking at the parent nodes of the currently considered leaf. If there exist a parent node ℓ_p reaching the current node ℓ with input j', a node ℓ' reachable from the parent node ℓ_p with list $j : j'$, and inputs j and j' are exchangeable, i.e., $j \overset{\downarrow}{\diamond} j'$, then the edge is drawn to the existing node ℓ'. This is depicted in Figure 2 (a), where the dashed edge is added. Otherwise, if one of the above conditions is violated (i.e., either the inputs cannot be exchanged or the node ℓ' has not been generated yet), a new node $\ell{+\!+}j$ is created and an edge drawn there. As an example, the case where for all $\ell_p \overset{j:j'}{\rightarrowtail}{}^*_i \ell'$ we have $j \not\overset{\downarrow}{\diamond} j'$ is depicted in Figure 2 (b). There, the dashed edge and the dashed node are added to the DAG. This process finishes at leaves

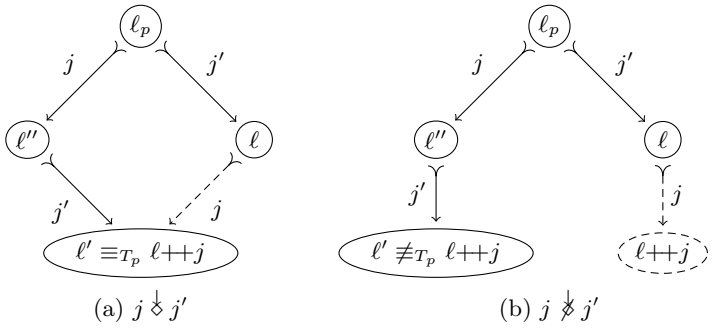


Fig. 2. Construction of the power-independence DAG

for which the list of considered inputs contains every input exactly once. It can furthermore be shown that the above construction always yields the power-independence DAG. Next, we prove that this DAG exactly distinguishes between the equivalence classes of the power-independence relation $\equiv_{T_p}$.

Theorem 7. *Let T_p be a power-extended vector-based transition system with m inputs and let $G_i = (V_i, \rightarrowtail_i)$ be its power-independence DAG.*

Then, for all orders $\pi_1, \pi_2 \in \Pi_m$, π_1 and π_2 are power-equivalent, iff there exist paths $\emptyset \overset{\pi_1}{\rightarrowtail}{}_i^* \pi$ and $\emptyset \overset{\pi_2}{\rightarrowtail}{}_i^* \pi$ in G_i for some order $\pi \in \Pi_m$.

Proof. To prove the "if" direction, we observe that due to the definition of the power-independence DAG in Definition 6, all nodes on a path, when appending the remaining considered inputs, are power-independent. This directly entails $\pi_1 \equiv_{T_p} \pi \equiv_{T_p} \pi_2$.

The "only-if" direction is proved by an induction over the number of swappings needed to reach π_1 from π_2, cf. Definition 4. If there are none, then $\pi_1 = \pi_2$ and the theorem trivially holds. Otherwise, we can apply the induction hypothesis to π_2 and the order π', resulting from π_1 by undoing the last swapping of j and $j{+}1$. This gives two paths $\emptyset \overset{\pi'}{\rightarrowtail}{}_i^* \pi$ and $\emptyset \overset{\pi_2}{\rightarrowtail}{}_i^* \pi$ in the graph. Since the two swapped positions j and $j+1$ are power-independent, and the rest of the orders π_1 and π' are the same, the two paths induced by these two orders must have the diamond shape due to the requirement in Definition 6, proving that also a path $\emptyset \overset{\pi_1}{\rightarrowtail}{}_i^* \pi$ exists. □

In summary, to determine the power-independent orders of a given power-extended vector-based transition system, we construct its power-independence DAG. Due to the above theorem, the lists contained in the leaves of the power-independence DAG are representatives of the different equivalence classes of orders that have to be considered for power characterization, i.e., only one of these orders has to be measured to obtain the real power consumption of all equivalent orders. Therefore, the number of leaves compared to the number of all possible orders is a measure for the reduction obtained by our method.

4 Selecting Orders to Minimize Power Consumption

The technique introduced in the previous section is useful in characterizing the power consumption of a cell. However, in order to minimize the power consumption, we develop a slightly different technique. Namely, contrary to the previous section, where we identify orders that always have the same dynamic power consumption, we now want to identify orders that are functionally independent (i.e., they do not influence the computation of a next state) but may have different power consumption. Then, by taking the order (one representative order among the equivalent orders) that consumes the least amount of power, the dynamic power consumption of computing the next state can be reduced.

For this purpose, we again define a DAG structure describing the different possible orders, but now we identify nodes that are computing the same next state, i.e., the inputs leading to such a shared node only need to have the diamond property regarding the functionality and not necessarily regarding the power consumption. Furthermore, we add to each node a back-pointer that determines which input leads to less power consumption. Then, by traversing the DAG from some leaf to the root following these back-pointers, one can construct the order that computes the same next state but uses minimal power. Next, we formalize this intuition.

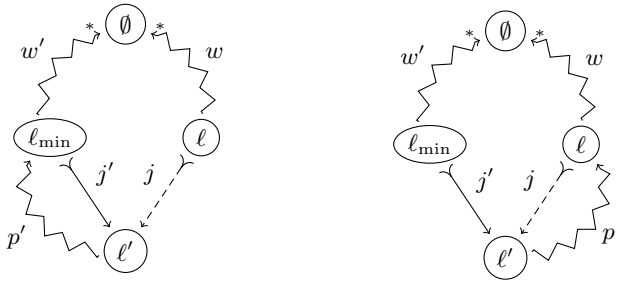
Definition 8. *Given a vector-based transition system $T = (S, I, \to)$ with m inputs and its power-extended vector-based transition system $T_p = (\mathbb{R} \times S, I, \to_p)$, we define the* power-sum DAG *$G_s = (V_s, \rightarrowtail_s, \rightsquigarrow_s)$, where $V_s \subseteq \mathcal{L}_m$, $\rightarrowtail_s \subseteq V_s \times V_s$, and $\rightsquigarrow_s \subseteq V_s \times (S \times I \times I) \times V_s$. The root is defined to be $\emptyset \in V_s$.*

The transition relation *$\rightarrowtail_s$ is defined for every $\ell \in V_s$ and every $1 \leq j \leq m$ as $\ell \rightarrowtail_s \ell'$ for some unique ℓ' such that $\ell +\!\!+ j \equiv_T \ell'$.*

The back-pointer relation *$\rightsquigarrow_s$ is defined for every $\ell \in V_s$, $s;\vec{i} \in S \times I$, and $\vec{i'} \in I$ as $\emptyset \overset{s;\vec{i};\vec{i'}}{\not\rightsquigarrow}_s \ell$ and, if $\ell \neq \emptyset$, $\ell \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \ell'$ for some unique $\ell' \in V_s$ with $\ell' \overset{j'}{\rightarrowtail}_s \ell$, $\ell' = j_1 : \cdots : j_h$, and $0;s;\vec{i} \xrightarrow{j_1}_p \ldots \xrightarrow{j_h}_p \circ \xrightarrow{j'}_p w;s';\vec{i'}$ for which $w \in \mathbb{R}$ is minimal.*

Note that in the definition of the transition relation of this DAG, we use independence based on equal states, not the extended power-independence which also checks for equal power consumption. Finally, we remark that again labels of edges are left out, but the transition relation $\rightarrowtail_s$ can be understood as labeled by an input position $1 \leq j \leq m$, indicating the added input coordinate that has been considered. This was already made use of in the definition of the back-pointer relation, but this position can again be recovered as the single input coordinate by which the two lists differ.

The construction of the power-sum DAG works similarly to the construction of the power-independence DAG. For it, we use the auxiliary function $w_{\min}$, which assigns to every node and state and input transition the minimal weight that the resulting state can be reached with, i.e., for $\ell \in V_s$ and $s;\vec{i};\vec{i'} \in S \times I \times I$, $w_{\min}(\ell, s;\vec{i};\vec{i'}) = w$ if $0;s;\vec{i} \overset{\ell'}{\twoheadrightarrow}_{T_p} w;s';\vec{i'}$ and w is minimal among all $\ell' \equiv_T \ell$. This



(a) Before adding $\ell \overset{j}{\rightarrowtail}_s \ell'$ (b) $w + p < w' + p'$

Fig. 3. Construction of the back-pointer relation in the power-sum DAG for some state and input vector transition $s;\vec{i};\vec{i'} \in S \times I \times I$

can be efficiently read from the back-pointer relation. To complete the function, we define $w_{\min}(\ell, s;\vec{i};\vec{i'}) = \infty$ if no $\ell' \in V_s$ exists such that $\ell \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \ell'$.

We start with the root node $\emptyset$ and add nodes in a breadth-first fashion. At each step, for each leaf ℓ of the DAG, we add an edge for every input position $1 \leq j \leq m$ that is not yet contained in ℓ. If there exists a node ℓ' such that $\ell_p \overset{j'}{\rightarrowtail}_s \ell$, $\ell_p \overset{j:j'}{\rightarrowtail}_s^* \ell'$, and $j \downarrow\!\!\diamond_T j'$, then the edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ is added. Otherwise, a new node $\ell{+}{+}j$ is added to the DAG, and the edge $\ell \overset{j}{\rightarrowtail}_s \ell{+}{+}j$ is added. This is the same construction that was used for the power-independence DAG G_i, illustrated in Figure 2, only here the commuting diamond property does not take the power consumption into account.

For the back-pointer relation, we use that sub-paths of a path with minimal weight also are of minimal weight, since otherwise a sub-path could be replaced by a smaller one. So, when adding an edge $\ell \overset{j}{\rightarrowtail}_s \ell'$, we define $\ell' \overset{s;\vec{i};\vec{i}}{\rightsquigarrow}_s \ell$ if $s;\vec{i} \overset{\ell}{\twoheadrightarrow} s_0;\vec{i}_0 \overset{j}{\rightarrow} s';\vec{i'}$ and $w_{\min}(\ell, s;\vec{i};\vec{i'}) + p(s_0, s') < w_{\min}(\ell', s;\vec{i};\vec{i'})$, otherwise we leave $\rightsquigarrow_s$ unchanged. Note that if ℓ' is a new node, then always the first case is applied, since the sum is always smaller than ∞.

An illustration of the back-pointer construction is shown in Figure 3, where the dashed edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ is to be added. Initially, we assume that there already is a node $\ell_{\min}$ such that $\ell' \overset{s;\vec{i};\vec{i}}{\rightsquigarrow}_s \ell_{\min}$, i.e., the power consumption is minimal if taking the minimal path from the root $\emptyset$ to node $\ell_{\min}$, which is assumed to have weight w', and then extending it by considering coordinate j', whose power consumption we assume to be p'. This situation is depicted in Figure 3 (a). Next, the node ℓ is considered. Note that $\ell' \equiv_T \ell{+}{+}j \equiv_T \ell_{\min}{+}{+}j'$, as otherwise the edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ would not be drawn. We assume the weight of the minimal path from the root $\emptyset$ to the node ℓ to be w and the power consumption of the step from ℓ to ℓ' to be p. In case $w + p < w' + p'$, then we have found a new minimal path for ℓ', thus we update the back-pointer relation as shown in Figure 3 (b). Otherwise, the previous path of the back-pointers is still giving the minimal path

even after adding $\ell \overset{j}{\rightarrowtail}_s \ell'$, so in that case the back-pointer relation remains as depicted in Figure 3 (a).

It can be shown that the above construction yields exactly the power-sum DAG G_s of a power-extended vector-based transition system. In the following theorem, it is shown that G_s identifies all orders that lead to the same state and a construction of the order consuming the minimal amount of power is given.

Theorem 9. *Let $T_p = (\mathbb{R}\times S, I, \rightarrow_p)$ be a power-extended vector-based transition system with m inputs and power-sum DAG $G_s = (V_s, \rightarrowtail_s, \rightsquigarrow_s)$. Furthermore, let $\pi, \pi' \in \Pi_m$ be some orders and $s;\vec{i};\vec{i'} \in S \times I \times I$ be some state together with previous and next input vectors.*

If $\emptyset \overset{\pi}{\rightarrowtail}_s^* \pi'$, then a path $\pi' = \ell_m \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \dots \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \ell_0 = \emptyset$ exists and $\pi \equiv_T \pi' \equiv_T \pi''$ for $\pi'' = j_1 : \dots : j_m \in \Pi_m$ defined by $\ell_r = \ell_{r-1} +\!\!+ j_r$ for all $1 \le r \le m$ such that $0;s;\vec{i} \overset{\pi''}{\twoheadrightarrow} w;s';\vec{i'}$ and w is minimal.

Proof. Existence of the back-pointer path and hence of π'' is guaranteed by the (unique) existence of a successor w.r.t. $\rightsquigarrow_s$ for every node that is not the root and since $\ell_r \neq \emptyset$ for every $1 \le r \le m$. The property $\pi \equiv_T \pi' \equiv_T \pi''$ directly follows from the definition of the transition relation of G_s. Finally, minimality of w follows from the definition of the back-pointer relation of G_s. □

Given a cell and its power-sum DAG G_s, one can obtain the order consuming the least amount of power for a given state, input vector transition, and order π in which the inputs are to be changed. This works by first traversing the DAG G_s according to the order π, which will result in a leaf π' of the DAG. From the leaf, the back pointer relation is followed upwards to the root, giving another order π'' with $\pi'' \equiv_T \pi' \equiv_T \pi$ that consumes the least amount of power, as shown in Theorem 9. Enforcing this order π'' can for example be done by adding delays, which is also proposed in [7].

5 Implementation

The techniques presented in Sections 3 and 4 were implemented in a prototype tool. This tool first parses a SPICE netlist and builds a symbolic vector-based transition system from it using the algorithm of [2], where states consist of a vector of formulas, computing values from the set $\{\mathsf{0}, \mathsf{1}, \mathsf{Z}\}$. The values $\mathsf{0}$ and $\mathsf{1}$ correspond to the logic values false and true, respectively, and represent an active path from a wire in the netlist to the low and high power rails, respectively. The third value, Z, represents a floating wire that has neither a path to the low nor to the high power rail. As the initial state of the netlist, we allow arbitrary values for all of the wires. The inputs are restricted to the binary values $\mathsf{0}$ and $\mathsf{1}$.

The power consumption of a transition is computed by the function p in Definition 1. In our implementation, this function is defined as $p(\vec{s}, \vec{s'}) = \left|\{1 \le j \le n : \vec{s}|_j = \mathsf{0}, \vec{s'}|_j = \mathsf{1}\}\right|$ for a netlist consisting of n wires, i.e., it counts the number of wires that transition from $\mathsf{0}$ to $\mathsf{1}$.

Building the power-independence DAG is then performed by first computing the diamond relation for all pairs of inputs (also taking power consumption into account). This is done symbolically using BDDs, requiring a total of $O(n \cdot m^2)$ BDD comparisons for m inputs and n state variables. Here, for every pair of input variables (of which there are $O(m^2)$ many) and every of the n state variables, two BDDs are constructed. The first computes the next state function after applying the two inputs in one order, the second BDD computes the next state function after applying the inputs in the other order. The currently considered pair of inputs has the power-extended diamond relation, if and only if these pairs of BDDs are equal for all state variables and the total number of wire chargings is the same. Finally, we construct the power-independence DAG as described in Section 3.2.

If the power-sum DAG is to be constructed, as described in Section 4, then we first need to compute the functional independence relation for all pairs of inputs. This also requires $O(n \cdot m^2)$ BDD comparisons. Furthermore, we need to keep track of the state to which a list of input coordinates leads, to be able to construct the back-pointer relation. For this purpose, we unroll the symbolic transition relation, i.e., we create a new transition relation that computes, given a starting state and input vector, the state and input vector after changing the inputs in the order of the currently considered node. This is used to create a symbolic formula computing the number of wires charged when adding another input to the list. Among these formulas we finally compute a symbolic minimum that indicates which parent node leads to minimal power consumption.

6 Experimental Results

We applied our technique to reduce the number of considered orders, which was presented in Section 3, and the technique to select an equivalent order that consumes less power, presented in Section 4, to the open-source Nangate Open Cell Library [4]. For each of the contained netlists, the SPICE source was parsed, a transition system created, and the power-independence DAG or power-sum DAG built and traversed to enumerate all equivalence classes. All of our experiments were conducted on a commodity PC equipped with an Intel Pentium 4 3.0 GHz processor and 1 GB RAM running Linux.

6.1 Reducing Input Vector Orders

Our results for reducing the number of considered orders with different functional or power consumption behavior are presented in Table 1, where the first column gives the name of the cell, the second column gives both the number of inputs and the number of wires, the third column the number of all possible orders, and the fourth column shows the number of different equivalence classes returned by our approach together with the time it took to compute these. Finally, the last column demonstrates the achievable power reduction, to be explained in Section 6.2.

For combinational cells, marked with "(c)" in Table 1, the results show that our approach cannot reduce the number of orders that have to be considered. This is usually due to situations in which wires are in one order first discharged only to be finally charged, whereas evaluating them in another order keeps the wire charged during the whole evaluation. Thus, all possible orders have to be considered during power characterization of these cells.

For sequential cells however, we can observe some larger savings especially for the larger cells. For example, in case of the largest cell in the library, the cell `SDFFRS`, we could reduce the number of orders to consider from 720 to only 288, which is a reduction by 60 %. Especially for sequential cells these savings have an effect, since for these cells the characterization not only has to take the possible input combinations into account, but also the current internal state. Overall, when summing up the absolute number of orders that have to be considered for the sequential cells, we get a reduction by more than 47 %. This is especially advantageous for the large cells, as witnessed by the average of the reduction rates of sequential cells, which is only slightly above 16 %. So especially for large sequential cells with lots of possible orders, our approach can reduce the number of orders that have to be considered significantly.

6.2 Selecting Input Vector Orders

We also evaluated the technique presented in Section 4, which computes the functionally equivalent orders and a path back that uses the minimal amount of power, using the open-source Nangate Open Cell Library [4]. The results are shown in the last column of Table 1, where the number of equivalence classes w.r.t. $\equiv_T$, the average number of maximal differences in wires chargings, and the amount of time for constructing the DAG and computing the result are given.

The results show that for combinational cells all orders lead to the same final state, which is expected as the state is completely determined by the new input values. For the sequential cells we observe that not all orders lead to the same final state, as there are multiple leaves in the power-sum DAG. This happens because the computation can depend on internally stored values, which might have different values when applying the input changes in different orders.

We illustrate the selection of orders by means of an example. Consider the scan logic of the cell `SDFFRS` (which is also the same in the cells beginning with `SDFF`), which is a multiplexer (mux) that selects, based on the value of the scan enable signal, between the data input and the scan input. In case the scan enable signal changes from `0` to `1` and the data input changes, then the power-sum DAG tells us that it is more power-efficient to first change the scan enable signal and then change the data input, than vice versa. This can be explained intuitively by the fact that while the scan enable signal is `0`, the mux is transparent to changes in the data input, so also wires connected to transistors controlled by the mux output are affected. This is not the case anymore if we first change the scan enable to `1`, so that the change in the data input cannot be observed at the output of the mux. In case of the cell `SDFFRS`, choosing the first order can cause 7 more wires to be charged.

Table 1. Results for the Nangate Open Cell Library

Cell	**#I / W**	**#Π_m**	**$\lvert G_i \rvert$ / t [s]**	**$\lvert G_s \rvert$: Avg / t [s]**
AND2 (c)	2 / 3	2	2 / 0.37	1 : 2.5 / 0.37
AND3 (c)	3 / 4	6	6 / 0.46	1 : 3.5 / 0.47
AND4 (c)	4 / 5	24	24 / 0.60	1 : 4.5 / 0.66
AOI211 (c)	4 / 4	24	24 / 0.62	1 : 4.0 / 0.64
AOI21 (c)	3 / 3	6	6 / 0.44	1 : 2.5 / 0.45
AOI221 (c)	5 / 5	120	120 / 0.97	1 : 7.0 / 1.14
AOI222 (c)	6 / 6	720	720 / 1.79	1 : 9.5 / 5.57
AOI22 (c)	4 / 4	24	24 / 0.68	1 : 5.0 / 0.66
BUF (c)	1 / 2	1	1 / 0.33	1 : 0.0 / 0.27
CLKBUF (c)	1 / 2	1	1 / 0.25	1 : 0.0 / 0.29
CLKGATETST	3 / 13	6	6 / 0.68	4 : 3.7 / 0.71
CLKGATE	2 / 11	2	2 / 0.54	2 : 0.0 / 0.54
DFFRS	4 / 24	24	24 / 1.20	12 : 1.1 / 1.82
DFFR	3 / 19	6	4 / 0.78	4 : 0.0 / 0.90
DFFS	3 / 19	6	6 / 0.82	4 : 1.0 / 0.90
DFF	2 / 16	2	2 / 0.63	2 : 0.0 / 0.68
DLH	2 / 9	2	2 / 0.50	2 : 0.0 / 0.52
DLL	2 / 9	2	2 / 0.53	2 : 0.0 / 0.52
FA (c)	3 / 14	6	6 / 0.71	1 : 3.0 / 0.76
HA (c)	2 / 8	2	2 / 0.46	1 : 1.0 / 0.48
INV (c)	1 / 1	1	1 / 0.24	1 : 0.0 / 0.25
MUX2 (c)	3 / 6	6	6 / 0.52	1 : 4.0 / 0.53
NAND2 (c)	2 / 2	2	2 / 0.35	1 : 1.5 / 0.35
NAND3 (c)	3 / 3	6	6 / 0.44	1 : 2.5 / 0.44
NAND4 (c)	4 / 4	24	24 / 0.58	1 : 3.5 / 0.61
NOR2 (c)	2 / 2	2	2 / 0.35	1 : 1.5 / 0.36
NOR3 (c)	3 / 3	6	6 / 0.47	1 : 2.5 / 0.45
NOR4 (c)	4 / 4	24	24 / 0.58	1 : 3.5 / 0.63
OAI211 (c)	4 / 4	24	24 / 0.59	1 : 4.0 / 0.64
OAI21 (c)	3 / 3	6	6 / 0.47	1 : 2.5 / 0.45
OAI221 (c)	5 / 5	120	120 / 1.07	1 : 7.0 / 1.22
OAI222 (c)	6 / 6	720	720 / 1.80	1 : 9.5 / 5.61
OAI22 (c)	4 / 4	24	24 / 0.62	1 : 5.0 / 0.66
OAI33 (c)	6 / 6	720	720 / 1.77	1 : 8.0 / 4.79
OR2 (c)	2 / 3	2	2 / 0.37	1 : 2.5 / 0.38
OR3 (c)	3 / 4	6	6 / 0.46	1 : 3.5 / 0.47
OR4 (c)	4 / 5	24	24 / 0.59	1 : 4.5 / 0.66
SDFFRS	6 / 30	720	288 / 2.99	48 : 6.4 / 13.01
SDFFR	5 / 25	120	96 / 1.61	16 : 6.4 / 3.07
SDFFS	5 / 25	120	36 / 1.49	16 : 6.0 / 2.77
SDFF	4 / 22	24	18 / 1.04	8 : 6.0 / 1.33
TBUF (c)	2 / 5	2	2 / 0.38	1 : 3.0 / 0.39
TINV (c)	2 / 4	2	2 / 0.39	1 : 3.0 / 0.38
TLAT	3 / 12	6	6 / 0.63	2 : 3.0 / 0.67
XNOR2 (c)	2 / 5	2	2 / 0.41	1 : 2.0 / 0.41
XOR2 (c)	2 / 5	2	2 / 0.44	1 : 2.0 / 0.40

Note that some correlation exists between the size of an equivalence class and the achievable power reduction: The more possible orders there are the more likely it is that another equivalent order with less power consumption exists. This can also be observed in results of Table 1, where the largest differences occur for combinational cells, which always have exactly one equivalence class.

7 Conclusions

This paper presented a technique to group together orders of applying input vectors which affect neither the functional behavior nor the power consumption. Such a technique is useful for power characterization, where it is sufficient to only choose one order of each of these equivalence classes, as all other elements exhibit the same behavior. Additionally, we presented a technique to select an order that uses the minimal power among functionally equivalent orders. Such a technique is useful when the order can be controlled, e.g., by means of the addition of delays as proposed in [7]. Then, one can force the evaluation to use an order consuming the minimal amount of power, without affecting the result of the computation. Both techniques were evaluated on the Nangate Open Cell Library and provided reductions within reasonable amounts of time.

Acknowledgments. We would like to thank (in alphabetical order) Alan Hu, Hamid Pourshaghaghi, and Shireesh Verma for their valuable input on this paper's topic. Also, we would like to thank the anonymous referees for their fruitful remarks.

References

1. Bogliolo, A., Benini, L., Ricco, B.: Power Estimation of Cell-Based CMOS Circuits. In: Proc. of DAC 1996, pp. 433–438. ACM, New York (1996)
2. Bryant, R.: Boolean Analysis of MOS Circuits. IEEE Transactions on Computer-Aided Design 6(4), 634–649 (1987)
3. Huang, M., Kwok, R., Chan, S.-P.: An Empirical Algorithm for Power Analysis in Deep Submicron Electronic Designs. VLSI Design 14(2), 219–227 (2000)
4. Nangate Inc. Open Cell Library v2008_10 SP1 (2008), `http://www.nangate.com/openlibrary/`
5. Raffelsieper, M., Mousavi, M.R., Roorda, J.-W., Strolenberg, C., Zantema, H.: Formal Analysis of Non-determinism in Verilog Cell Library Simulation Models. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 133–148. Springer, Heidelberg (2009)
6. Raffelsieper, M., Mousavi, M.R., Zantema, H.: Order-Independence of Vector-Based Transition Systems. In: Proc. of ACSD 2010, pp. 115–123. IEEE, Los Alamitos (2010)
7. Raghunathan, A., Dey, S., Jha, N.K.: Glitch Analysis and Reduction in Register Transfer Level Power Optimization. In: Proc. of DAC 1996, pp. 331–336. ACM, New York (1996)
8. Shen, W.-Z., Lin, J.-Y., Lu, J.-M.: CB-Power: A Hierarchical Cell-Based Power Characterization and Estimation Environment for Static CMOS Circuits. In: Proc. of ASP-DAC 1997, pp. 189–194. IEEE, Los Alamitos (1997)

An Automated Semantic-Based Approach for Creating Tasks from Matlab Simulink Models*

Matthias Büker[1], Werner Damm[2], Günter Ehmen[2], and Ingo Stierand[2]

[1] OFFIS - Institute for Computer Science, Oldenburg, Germany
matthias.bueker@offis.de
[2] University of Oldenburg, Germany
{damm,ehmen,stierand}@informatik.uni-oldenburg.de

Abstract. The approach proposed in this paper forms the front-end of a framework for the complete design flow from specification models of new automotive functions captured in Matlab Simulink to their distributed execution on hierarchical bus-based electronic architectures hosting the release of already deployed automotive functions. The process starts by deriving a task structure from a given Matlab Simulink model. Because the obtained network is typically unbalanced in the sense of computational node weights, nodes are melted following an optimization metric called *cohesion* where nodes are attracted by high communication density and repelled by high node weights. This reduces task-switching times by avoiding too lightweight tasks and relieves the bus by keeping inter-task communication low. This so-called *Task Creation* encloses the translation of the synchronous block diagram model of Simulink into a message-based task network formalism that serves as semantic base.

Keywords: simulink, creating tasks, design flow, distributed systems.

1 Introduction

We propose a framework that aims at automating significant parts of the design flow in the following typical scenario for embedded application development in automotive: given the electronic architecture A of a particular car model, we are looking for a conservative cost-optimized extension of this architecture to implement a new customer feature F. We consider typical hierarchical bus-based target architectures with a backbone TDMA based bus. In this work, we concentrate on the front-end of this framework where we assume that F is given as a Matlab Simulink model and propose an automated process to derive a balanced task structure serving as input to the design space exploration process. The goal is to optimize the granularity of the task structure while maintaining causality constraints by balancing computational load and minimizing communication density. This is achieved by introducing a metric called *cohesion* which reduces task-switching times by avoiding too lightweight tasks and relieves the

* This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 149–164, 2011.

bus by keeping inter-task communication low. As an interface between this front-end process and the design space exploration process, we use function networks, which provide a formal semantic base expressive enough to represent different communication and execution paradigms and all timing related aspects [4].

To formally represent Matlab Simulink models we follow Lublinerman et al. [12] where these models are defined as timed synchronous block diagrams (TBD). A related approach of Tripakis et al. [5] is also based on TBDs and translates Simulink models to Lustre to partition the generated code into modules that are executed on different processors communicating via a time-triggered bus. Contrary to our work, the focus lies on separating the code into different modules respecting a global partial order, while still performing a scheduling analysis for user-specified timing constraints. Producing modular sequential code from synchronous data-flow networks is also addressed by Pouzet et al. [13]. They decompose a given system into a minimum number of classes executed atomically and statically scheduled without restricting possible feedback loops between input and output. However, the question of efficient and modular code generation lies beyond our approach but can be esteemed as supplementary. Di Natale et al. propose [6] an optimization of the multitask implementation of Simulink models with real-time constraints on a single processor. The optimization goal is to reduce the use of rate transition blocks between different synchronous sets (that are connected blocks with the same sample time) to minimize buffering and latencies. The tasks for the scheduling analysis are determined by the synchronous sets while task priorities and execution order of function blocks within a task are optimized. Another work from Kugele et al. [11] is also based on synchronous languages and presents a way to deploy clusters - that are actually tasks - specified by the COLA language on a multi-processor platform. This allocation process is completed by a scheduling-analysis involving address generation and estimation of memory requirements for a pre-defined middle-ware. In this process they also rise the question of how to generate clusters of nodes (tasks) but currently assume that this is a decision that is taken manually by the user.

The contribution of this paper is as follows. Taking a translation scheme for Matlab Simulink models to function networks, we define a cohesion metric and an algorithm that partitions the resulting nodes to obtain a balanced task set with respect to computational weights, and also minimized communication demand between tasks. To obtain tasks correctly, we define formal composition operations for nodes in a function network and show semantics preservation of these operations in terms of causality (ordering) of node executions and timing.

Outline. We start in Section 2 with the definition of an extended task network formalism called *function network* as semantic base for the whole process followed by a short introduction into the translation concept from Matlab Simulink to function networks in Section 3. The actual approach of task creation including its semantics and application is presented in Section 4 completed by evaluation results of this approach in Section 5. Section 6 concludes the paper.

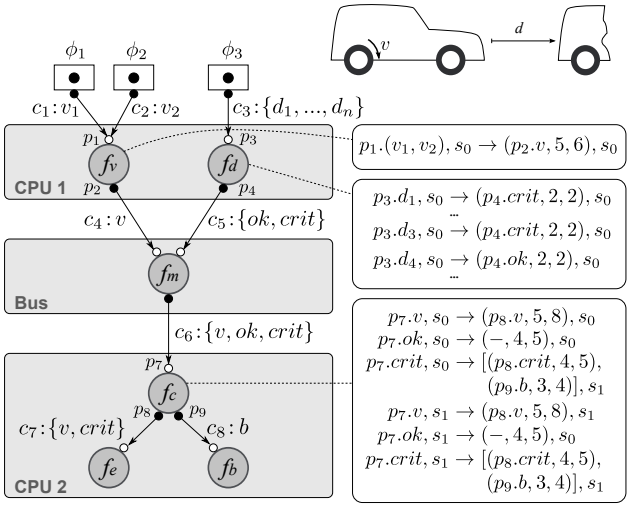


Fig. 1. Adaptive Cruise Control System

2 Function Networks

Like other task network formalisms, function networks [4] are directed graphs where nodes represent processing elements (tasks), and edges represent channels transmitting events between nodes. A *channel* may transport different events. Channel c_3 in Figure 1 for example transmits events from the set $\{d_1, ..., d_n\}$.

Event sources allow to model events sent by the environment to the network. In Figure 1, they are depicted as rectangles with filled circles. Source ϕ_3 represents for example a distance sensor delivering values $d_1, ..., d_n$. The occurrence of events is defined in terms of event streams [14]. Common streams define for example periodic or sporadic event occurrences. Event streams for function networks are defined by a tuple $(\Sigma^{out}, P^-, P^+, J, O)$ where Σ^{out} is a set of events, $[P^-, P^+]$ defines an interval of a minimum and maximum period for event occurrences, J is the jitter, and O is an initial offset.

The connection between nodes and channels is realized by ports. Activation of nodes is captured by their input ports (small white circles). An input port activates a node when at least one event has occurred at each incoming channel of that port. Node f_v in Figure 1 for example is activated when both an event v_1 on channel c_1 and v_2 on c_2 occurs. A node having multiple input ports is activated on the activation of any of its input ports. Combining multiple ports and multiple input channels allows modeling of complex node activations.

Function nodes employ internal state-transition systems to model for example functions that are sensitive to incoming events, and data access to, e.g. shared variables and FIFO buffers. Node f_c for example sends a braking event b whenever it is activated and the last captured distance was critical ($crit$). Each activation causes a delay for processing, depending on the input event, the current state, and the particular output port. Delays are taken from intervals

with best-case and worst-case bounds. For example, an event *crit* that activates node f_c in state s_0 needs between 3 and 4 ms to be sent to port p_8.

To simplify modeling data flow, the function network formalism is extended by data nodes, that are special function nodes modeling explicit data storage. We define different types of data nodes as persistent ones like *Shared* variables and *FIFO* buffers, and volatile ones like *Signals*. Another data node type is the *finite source* (*FSource*) producing an initial event at its output port at system startup while emitting the next event not before an event was received at any input port. This node type is used to model cycles in function networks. Its semantics is very similar to pre-allocated events in task networks with cyclic dependencies [9]. A further enrichment is the introduction of a new channel type named *read* channel. While common (*activation*) channels model control flow and cause an activation at their target function node, read channels model data dependencies, that is, reading access by a function node to a data node at the activation of that function node. Read channels are depicted as dotted arcs.

Definition 1 (Function Network). *A function network is a tuple* $FN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *where:*

- Σ *is a finite alphabet. Events are tuples* $e = (\sigma_1, \ldots, \sigma_k)$ *with* $\sigma_i \in \Sigma$. $\mathring{\Sigma} \subset \Sigma^*$ *is a finite set of events.*
- $\mathcal{P} = \mathcal{P}^I \dot\cup \mathcal{P}^O$ *is a set of input and output ports,*
- $\mathcal{C} = \mathcal{C}^A \cup \mathcal{C}^R \subseteq (\mathcal{P} \times \mathbb{N}^+ \times \mathbb{N}^+ \times \mathcal{P})$ *is a set of channels* $c = (p^{out},\ \delta,\ p^{in})$ *where* $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ *is a delay interval,* $c \in \mathcal{C}^A$ *are* activation *channels and* $c \in \mathcal{C}^R$ *are* read *channels leading exclusively from data to function nodes.*
- Φ *is a set of sources* $\phi = (EP, P^{out})$ *where* $EP = (\Sigma^{out}, P^-, P^+, J, O)$ *is an event pattern,* $\Sigma^{out} \subseteq \mathring{\Sigma}$ *are the events transmitted by the source.* $P^{out} \subseteq \mathcal{P}^O$ *is a set of output ports.*
- $\mathcal{F}$ *is a set of function nodes* $f = (P^{in}, \mathcal{A}, P^{out})$ *where* $P^{in} \subseteq \mathcal{P}^I$ *is a set of input ports, and* $P^{out} \subseteq \mathcal{P}^O$ *is a set of output ports.* $\mathcal{A} = (S, s_0, T)$ *is a timed transition system where* S *is a non-empty finite set of states,* $s_0 \in S$ *is the initial state, and* T *is a transition function*

$$T : P^{in} \times \mathring{\Sigma} \times S \to ((P^{out} \times \mathring{\Sigma}) \cup \{\bot\} \to \mathbb{N}^+ \times \mathbb{N}^+) \times S$$

  mapping combinations of ports, incoming events and states to ports, delay intervals and successor states. Symbol $\bot$ *denotes "no output port".*
- $\mathcal{D}$ *is a set of data nodes* $d = (\mathcal{P}^{in}, \delta, b, \mathcal{P}^{out})$ *where* $\mathcal{P}^{in} \subseteq \mathcal{P}^I$ *is a set of input ports,* $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ *is a delay interval,* $b \in \{FIFO, Shared, Signal, FSource\}$ *is a data node type.* $\mathcal{P}^{out} \subseteq \mathcal{P}^O$ *is a set of output ports.* ◇

3 From Specification Models to Function Networks

In our design process, function networks are obtained from a Matlab Simulink model by a set of transformation rules. Blocks are translated to function nodes and signals are translated to channels. The main challenge is here to translate a synchronous specification into a message-based task model. This is realized

by deriving a semantics for Matlab Simulink models based on updates of signal values and relating this to events in the function network. For signals between blocks of different *synchronous sets* (connected blocks with same sample time [6]) the Matlab Simulink concept of *Rate Transition Blocks* is mapped to function networks. Timing information is obtained by deriving event patterns for specific nodes in the function network. The behavior of the specification is translated into executable code by employing (existing) code generators such as Realtime Workshop or TargetLink. Worst case execution times (WCETs) are calculated for the resulting code blocks to assign weights to the respective function nodes. For this, we use a combination of well known techniques like static program analysis [7] and measuring. For translation, we pick up the basic idea of [12] where Matlab Simulink models are defined as *Timed Synchronous Block Diagrams* (TBDs).

3.1 Translating Simulink

The TBD is flattened as described by Lublinerman [12] while the hierarchy of subsystems is used to generate constrains for the task creation (see Section 4). As in [12], we assume the TBDs to be acyclic in that all cycles must contain at least one Moore-sequential block such as a "Unit Delay" block. In the following we shortly introduce the translation concept. More details can be found in [3].

Blocks. Each block b with n inputs and m outputs is translated to a function node f_b with one input port and m output ports. All input channels are synchronized at the input port to activate the function node only when all needed inputs have been computed i.e. the appendant events have been received. The delay of each transition is defined by determining the worst case execution time of this block assuming the input configuration described by the events.

Each *Moore-sequential* block b with n input signals and m output signals is translated to an *FSource* data node with one input port and m output ports. Each *Rate Transition Block* with an input signal from a block a and an output signal to a block b is translated to a special function node f_{rt} with a *Shared* data node converting the sample time of block a to the sample time of block b. *Data Store Memory* blocks and *Sink* blocks are translated to *Shared* data nodes.

Triggers and Signals. A *trigger* with a sample time ST connected to a block b with no input signals is translated to a *Source* with an event pattern implementing ST and an activation channel to the input port of f_b. A trigger t leading from port o of block a to block b is translated to an activation channel from the respective port p_o of f_a to the input port of f_b.

A *signal* x leading from output port o of block a to an input port of block b is translated as follows: If a and b have the same sample time x is translated to an activation channel from p_o^{out} to p_b^{in}. If a and b have different sample times we create a "virtual" rate transition block between a and b whose translation was described before. Additionally, we define the data size $DataSize(c)$ for each created channel c by considering the data type of the appendant Simulink signal. This is used to define weights in the subsequent task creation process.

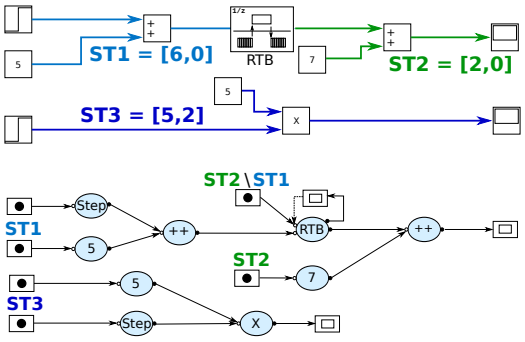


Fig. 2. Preserving Synchronous Sets of Matlab Simulink Models

3.2 Preserving Semantics

For both translation and the following task creation, preserving semantics of the original specification is a key issue. Matlab Simulink models are inherently untimed, where block executions and communication are instantaneously, and are not affected by delays and jitter due to variances in the delays. Obviously, nothing of this holds for any implementation of a Matlab Simulink model that runs on real hardware. Moreover, TBD models follow the synchronous paradigm while function networks are asynchronous models based on communication by message transfer. If we translate one paradigm to the other, we have to take care that ordering of block executions is maintained and that for each execution the currently available inputs match those of the original semantics.

The translation maintains the partial order of blocks induced by the structure of a TBD [6] and preserves the order of signal updates by corresponding events in the function network. For blocks of one synchronous set all input channels of the corresponding function node are synchronized. Each time a block is executed, its output signals are updated which is represented in the function node by producing an event on each output port for each activation. Between different synchronous sets a function node acts as rate transition translating from one sample time to the other. On top of Figure 2 a Simulink example is depicted with its function network translation beneath. The block RTB represents a rate transition leading from sample time ST_1=[6,0] to ST_2=[2,0]. In the function network this is realized by a function node RTB with a source implementing $ST_2 \setminus ST_1 = \{[6,2],[6,4]\}$ and a *Shared* data node storing input events.

To capture Matlab Simulink semantics correctly when the corresponding implementation is executed on a platform, we additionally have to ensure that all functions of a synchronous set are executed within their associated sample time. Thus, we define (causal) deadlines from the activation of any start node to the finished execution of any end node of a synchronous set. The length of the deadline is defined by the period of the set. For connected synchronous sets we need to define further deadlines over a complete path from a source to a sink while their length is determined by the minimum period of any set of the path. These

deadlines may also be non-causal depending on the relation of the involved periods. For example, in Figure 2 there exist deadlines for each synchronous set (of length 6, 2 and 5) and additional deadlines (which are causal here) over the sets with ST_1 and ST_2 with a length of 2. For the following task creation process it is important to maintain the synchronous sets because otherwise no valid deadlines could be defined. More details on perserving semantics can be found in [3].

4 Task Creation - Semantics and Application

A function network that results from translation of a Simulink model is typically unbalanced, consisting of a large number of nodes with high variance in computational node weights which estimate the load a node potentially produces on an ECU. If we treat each node as a single task, this would result in a large communication overhead when many lightweight tasks are spread over the distributed hardware resources in the pursuing design exploration process. Accordingly, we want to obtain a more suitable task set by merging function nodes into tasks.

The proceeding of Realtime Workshop is to put all blocks with the same sample time into one task. These may not only be blocks of the same synchronous set but also of independent sets sharing the same sample time which may lead to very "heavy" tasks. For the execution on a distributed real-time system, this strategy precludes any of these blocks from being executed concurrently, which increases the risk of deadline violations. Nevertheless, nodes of the same synchronous set are still good candidates to be executed in the same task [6]. But due to the possibly high variety of the number and weights of nodes in synchronous sets not all sets would necessarily result in useful tasks. For example, we do not want to allow arbitrary large weights because those tasks may be either not executable on some ECUs, or they would reduce the number of possible schedules due to large blocking times. On the other hand, tasks should not be too lightweight, because the sum of task switching times would increase and waste a significant amount of ECU capacity. From the perspective of the design space exploration, it is desirable to have tasks with balanced weights. This would largely reduce the impact of computational density of tasks, and the decision where to allocate a task would be more driven by the actual optimization criteria.

Another important issue for task creation is the communication between tasks which may get very expensive if tasks are mapped to different ECUs and a bus has to be used. A bus is not only comparably slow, but also often the bottleneck of such systems and can hardly be upgraded. Hence, another objective for task creation should be to minimize communication between tasks to relieve the bus. To achieve all of this, we introduce in the next section a metric called *cohesion*.

4.1 Cohesion and Weights

Formally, task creation partititions the set of function and data nodes (which are special function nodes) $\mathcal{N} = \mathcal{F} \cup \mathcal{D}$ into a task set $\mathcal{T} = \{\tau_1, ..., \tau_m\}$ where $\tau_i = \{n_{i,1}, ..., n_{i,k}\}$, $n_{i,j} \in \mathcal{N}$. The communication structure of the resulting

task set is determined by the set of channels $\mathcal{C}(\mathcal{T})$ between different partitions. The task set shall be chosen such that communication density is minimized and node weights are balanced. Node balancing is achieved by minimizing the standard deviation with respect to the aspired task weight leading to preferably merging nodes with low weights, while communication is minimized by reducing the weight of $\mathcal{C}(\mathcal{T})$. For the definition of cohesion, we introduce weight factors $\alpha, \beta > 0$ that are adjusted by user preference to control the process. Furthermore, we assume m^- to be the desired minimum number of tasks which also determines the aspired task weight. This leads to the following definition of *cohesion*:

$$
\begin{aligned}
\mathit{cohesion}(FN, \mathcal{T}) &= \alpha \cdot \widehat{w}(\mathcal{T}) + \beta \cdot \mathit{com}(\mathcal{C}(\mathcal{T}))\,, \quad \text{where} \\
\widehat{w}(\mathcal{T}) &= 1/m \cdot \sqrt{\sum\nolimits_{i=1}^{m} (\overline{w} - w(\tau_i))^2} \quad \text{(standard deviation)} \\
\overline{w} &= 1/m^- \cdot \sum\nolimits_{n \in \mathcal{N}} w(n) \quad \text{(aspired task weight)} \\
w(\tau_i) &= \sum\nolimits_{n_{i,j} \in \tau_i} w(n_{i,j}) \quad \text{(weight of task } \tau_i\text{)} \\
\mathit{com}(\mathcal{C}(\mathcal{T})) &= \sum\nolimits_{c \in \mathcal{C}(\mathcal{T})} \mathit{com}(c) \quad \text{(sum of communication weights)}
\end{aligned}
$$

The weight $w(n)$ of a node depends on its execution times in terms of transition delays and its activation pattern while execution times strongly depend on the compiler target. We define the delay of a transition as the minimum WCET among all potential processors of the target architecture, because without further knowledge about possible allocations we assume that each node will be allocated to its best fitting processor. More precisely, the weight of a node is defined as the sum of its port weights. The weight of a port is the maximum delay of all transitions starting at this port divided by the ports lower period bound. The period depends on the event pattern of preceding nodes and event sources. The weight for function node $f = (P^{in}, \mathcal{A}, P^{out})$ is defined as:

$$
w(f) = \sum\nolimits_{p_i \in P^{in}} \left(\frac{1}{P_i^-} \cdot \max\nolimits_{t_{i,j}} (\Delta^+(t_{i,j}))\right), \quad \text{where}
$$

$\Delta^+(t_{i,j})$ is the upper delay bound of the jth transition starting from input port p_i and P_i^- is the lower period bound of p_i. Communication density is defined in terms of weights for channels c depending on their data size, the communication rate, and the maximum bandwidth in $bytes/s$ of all buses k. It is defined as:

$$
\mathit{com}(c) = \frac{\mathit{DataSize}(c)}{\max \mathit{Bandwidth}_k} \cdot \frac{1}{P_c^-}\,, \quad \text{where}
$$

$\mathit{DataSize}(c)$ is the data size of channel c and P_c^- is the lower period bound of c. The period of a channel can in general be retrieved for example by so-called event stream propagation for task networks. For Matlab Simulink models, the period of any communication and any port activation is always well-defined.

Beside the optimization goal of minimizing the cohesion function, there exists a set of user-controlled constraints restricting the task creation process. First, we introduce minimum and maximum achievable task weights. The minimum task weight w^- is intended to counteract thrashing caused by tasks with too small

execution time, which induces frequent task switching and thus lowers processor utilization. The maximum task weight w^+ describes the maximum utilization a single task should involve on a processor and ensures that there is sufficient potential parallelism, thus allowing to reduce end-to-end latencies. As α and β, these parameters will typically highly depend on the respective application. Second, further constraints can be obtained from the hierarchical structure of the original specification model. For example, in Simulink the user may claim that all blocks of a certain subsystem have to be mapped into the same task.

4.2 Syntax and Semantics of Task Creation

In this section, we elaborate on the question what task creation actually means and which semantic consequences it implicates. The process of task creation is divided into three independent operations: merging of function nodes, elimination of local data nodes and elimination of self-activations. The first operation is mandatory for task creation while the other operations are optional. The operations are defined with the help of a component concept where a component is a part of a function network with a well-defined interface of ports to the remaining network. Each operation replaces one component by another one with the same interface. For semantic correctness of an operation, the causality i.e. the partial order of interface events has to be maintained. This also holds for internal events as long as they remain observable when applying the operation. If an event is no longer observable, causality is preserved by maintaining the control flow.

Merging nodes. When two function nodes are merged this involves a restructuring of the function network by replacing a component of two function nodes f_1 and f_2 by a component with one function node f_{1+2} with the same interface. This means that each the sets of input ports and output ports of f_1 and f_2 are unified. The transition system of f_{1+2} is obtained by building the usual parallel composition $\|$ [1] of the transition systems of f_1 and f_2.

Definition 2 (Node Merging). *Let* $FN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be a function network and* $f_1 = (P_1^{in}, \mathcal{A}_1, P_1^{out}) \in \mathcal{F}$ *and* $f_2 = (P_2^{in}, \mathcal{A}_2, P_2^{out}) \in \mathcal{F}$ *be two function nodes. The merge operation is defined as follows:*
$merge(FN, f_1, f_2) = (\Sigma, \mathcal{P}, (\mathcal{F} \setminus \{f_1, f_2\}) \cup \{f_{1+2}\}, \Phi, \mathcal{D}, \mathcal{C})$*, where* $f_{1+2} = (\mathcal{P}_1^{in} \cup \mathcal{P}_2^{in}, \mathcal{A}_1 \parallel \mathcal{A}_2, \mathcal{P}_1^{out} \cup \mathcal{P}_2^{out})$ ◇

The semantic consequences of merging two function nodes f_1 and f_2 is mainly that f_1 and f_2 are now executed on the same scheduling resource i.e. transitions of f_1 and f_2 cannot be executed concurrently anymore. But even though we change function network behavior by this operation causality is still preserved in terms of partial ordering of events. This is because all events, ports, channels and data nodes are maintained as well as the transition systems of the original function nodes. Therefore, also the partial order of all events (including interface events) is preserved. Concerning timing, node merging may enlarge the delay between the arrival of an event at an input port and the emitted output event, because transitions that could be executed concurrently before cannot be

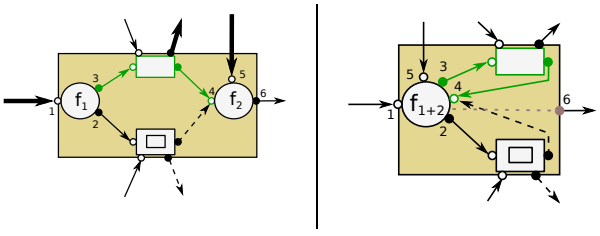


Fig. 3. Merging two function nodes (left side) into one (right side)

executed concurrently after merging. Because computational weights of function nodes are the sum of their port weights and all ports are maintained including their transitions, the weight of f_{1+2} is the sum of the single weights of f_1 and f_2 as claimed in the weight calculation.

In Figure 3 on the left side a component of a function network with two function nodes f_1 and f_2 is depicted where f_1 triggers f_2 via a *Signal* node and two activation channels. Furthermore, there are read and activation channels to a *Shared* data node. The same function network part after merging f_1 and f_2 is depicted on the right side of Figure 3. The activation path is now a self-activation i.e. f_{1+2} activates itself at a different input port. The *Shared* data node is unaffected and the read channel moves with its target port to f_{1+2}.

The merging operation is associative because both the joining of ports and the parallel composition of transition systems is associative. This becomes important for the application of this operation in the task creation algorithm. This operation can also be applied for two function nodes that are directly connected by an activation channel (with a delay > 0).

Elimination of Local Data Nodes. A data node d is local if it is exclusively connected to a function node f *and* in the same task partition as f. When eliminating a data node, also the corresponding read and write channels are removed. The transition system of f is modified such that the respective events are removed from any transition. Additionally, the output port p_w writing to d is removed and each transition that emitted events at p_w now writes $\perp$ instead.

Definition 3 (Data Node Elimination). *Let* $FN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ *be a function network,* $f = (\mathcal{P}^{in}, (S, s_0, T), \mathcal{P}^{out}) \in \mathcal{F}$ *a function node and* $d \in \mathcal{D}$ *a data node with an incoming activation channel* $c_w = (p_w, \delta_w, p_d)$ *(*$p_w \in \mathcal{P}^{out}$*) transmitting event* w *and an outgoing read channel* $c_r = (p_{d'}, \delta_r, p_r)$ *(*$p_r \in \mathcal{P}^{in}$*) transmitting event* r*. The data node elimination function is defined as follows:*

$$elim_d(FN, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}'), \text{ where}$$

- $\Sigma' = \Sigma \setminus \{r, w\}$, $\mathcal{P}' = \mathcal{P} \setminus \{p_d, p_{d'}, p_w\}$,
- $\mathcal{D}' = \mathcal{D} \setminus \{d\}$, $\mathcal{C}' = \mathcal{C} \setminus \{c_r, c_w\}$ *and*
- $f = (\mathcal{P}^{in}, (S, s_0, T'), \mathcal{P}^{out} \setminus \{p_w\})$, T' *contains all transitions from* T *while*
  - *each occurence of* p_w *in a transition is replaced by* $\perp$,
  - *event* r *is deleted in each transition where* E *contains* r. ◇

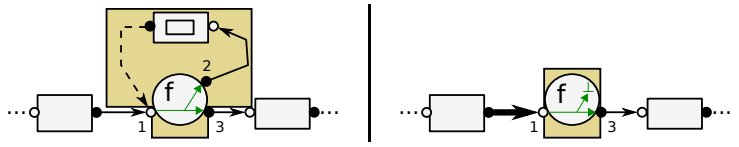


Fig. 4. Elimination of a local data node

On the assumption that the behavior of the function node does not depend on the read event of the removed data node, this operation maintains the causality of the remaining events. All input ports of the function node are obtained together with all activation events of that node. The transitions of the function node are maintained as well while they are cleaned by the read event r. Thus, the partial order between input and output events of the components interface is still valid. Concerning timing, the delay between any input and output signal that involves the reading of event r becomes smaller because the data is now available locally and the time for reading the event (possibly over a bus) is saved. Thus, any end-to-end deadline that was valid before this operation is still valid after it. In Figure 4 on the left side a component is shown with a function node and a local data node that is eliminated on the right side. The arrows in the function node indicate the affected transitions to show that these are maintained even if an output port is removed.

Elimination of Self-Activations. Self-activations are self-loops of a function node f either via a *Signal* node or an activation channel with a delay > 0. They particularly arise when two function nodes with an activation dependency are merged. Thus, their elimination is a typical continuation of the node melting process. The consequences of the elimination of self-activations are, that an involved data node is removed if it is not used by other function nodes. Additionally, channels may be deleted including the appendant ports and events.

To be able to apply this operation without violating causality of events, the input port of the self-activation loop must not have any other incoming channels. A further necessary condition for a loop containing a data node d is, that d must not have *both* incoming and outgoing channels to other function nodes than f. In this case, it would not be possible to remove the self-activation without affecting activations from or to other nodes. Before defining the operation for eliminating self-activations itself, we need to define some help functions. The first one adds an output delay to a given set of output specifications of a transition. An output specification is a pair of a port and event mapped to a delay interval.

Definition 4 (Output Delay Addition). *Let* $\psi = \{(p'_1, E'_1 \rightarrow \delta_1), ..., (p'_n, E'_n \rightarrow \delta_n)\}$ *with* $\delta_i = [\delta_i^-, \delta_i^+]$ *be a set of output specifications, and* $\delta = [\delta^-, \delta^+]$ *a delay interval. Output Delay Addition is defined as:*
$\delta_{add}(\Psi, \delta) = \{(p'_1, E'_1 \rightarrow (\delta_1 + \delta)), ..., (p'_n, E'_n \rightarrow (\delta_n + \delta))\}$,
where $\delta_i + \delta \equiv [\delta_i^- + \delta^-, \delta_i^+ + \delta^+]$ ◇

Next, we define how a given transition system changes when a self-activation via an output port p_w and an input port p_a is eliminated. For each transition

that does not contain one of these two ports nothing changes. But all pairs of transitions that would execute successive in the case of a self-activation need to be concatenated. This means, that the left part (input port, input event, origin state) of the first transition becomes also the left part for the concatenated transition. The right part of this new transition must not fire events at p_w, if this port is removed during self-loop elimination. Instead, the delays of the affected events are added to the delays of all output ports of the second transition. All other output specifications remain unchanged.

Definition 5 (Self-Transition Concatenation). *Let T be a transition system, p_a be an input port and p_w an output port of a self-activation. Self-Transition Concatenation is defined as: $concat(T, p_a, p_w) = T'$ where*

1. $\forall t = (p,\ E,\ s \rightarrow \Psi,\ s') \in T,\ p \neq p_a,\ \nexists\ \psi = (p_w, E_w \rightarrow \delta_w) \in \Psi$ *holds:* $t \in T'$
2. *For each pair of transitions*
   - $t_1 = (p_1,\ E_1,\ s_1 \rightarrow \Psi_1,\ s_1') \in T$ *where* $\exists \psi_w = (p_w, E_w \rightarrow \delta_w) \in \Psi_1$ *and*
   - $t_2 = (p_a, E_2, s_2 \rightarrow \Psi_2, s_2') \in T$ *holds:*

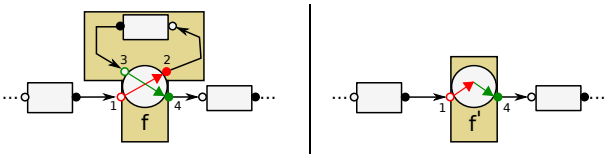
   $\exists t \in T'$ *with* $t = (p_1,\ E_1,\ s_1 \rightarrow \Psi_1 \setminus \psi_w \cup \delta_{add}(\Psi_2, \delta_w, s_2'))$ ◇

Elimination of self-activations is defined for a function node f that activates itself via a *Signal* node d. This is the more general case compared to activations by direct activation channels which are covered as well as a simplification of case 1) of the subsequent definition. A self-activation is resolved by replacing it by a set of concatenated transitions. This means that succeeding executions of the self-activation are merged into one using the previously defined functions.

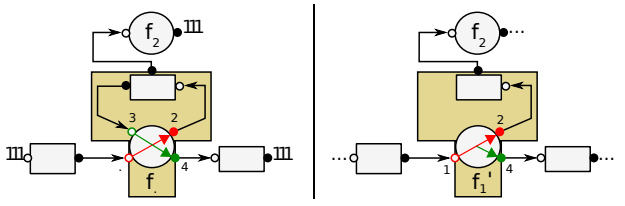
Definition 6 (Self-Activation Elimination). *Let $FN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network, $f = (\mathcal{P}^{in}, (S, s_0, T), \mathcal{P}^{out}) \in \mathcal{F}$ a function node and $d = (\mathcal{P}_d^{in}, \delta_d, b_d, \mathcal{P}_d^{out}) \in \mathcal{D}$ a Signal data node that has an incoming activation channel $c_w = (p_w, \delta_w, p_d)$ ($p_w \in \mathcal{P}^{out}$) transmitting event w and an outgoing activation channel $c_a = (p_{d'}, \delta_a, p_a)$ ($p_a \in \mathcal{P}^{in}$) to f transmitting event a. Self-activation elimination is defined as follows:*

$elim_a(FN, f, d) = (\Sigma', \mathcal{P}', \mathcal{F}, \Phi, \mathcal{D}', \mathcal{C}')$ *while we distinguish the following cases:*

1. *If d has no other channels than c_w and c_a, then*
   - $\Sigma' = \Sigma \setminus \{w, a\}$, $\mathcal{P}' = \mathcal{P} \setminus \{p_d, p_{d'}, p_a, p_w\}$, $\mathcal{D}' = \mathcal{D} \setminus d$, $\mathcal{C}' = \mathcal{C} \setminus \{c_w, c_a\}$,
   - $f = (\mathcal{P}^{in} \setminus \{p_a\}, (S, s_0, concat(T, p_a, p_w)), \mathcal{P}^{out} \setminus \{p_w\})$
2. *If d has an additional activation channel to another function node, then*
   - $\Sigma' = \Sigma \setminus \{a\}$, $\mathcal{P}' = \mathcal{P} \setminus \{p_{d'}, p_a\}$, $\mathcal{D}' = \mathcal{D}$, $\mathcal{C}' = \mathcal{C} \setminus c_a$,
   - $f = (\mathcal{P}^{in} \setminus \{p_a\}, (S, s_0, concat(T, p_a, p_w) \cup T_w), \mathcal{P}^{out})$
     where $T_w = \{(p,\ E,\ s \rightarrow \Psi,\ s') \in T \mid \exists (p_w, E_w \rightarrow \delta_w) \in \Psi\}$
   - $d = (\mathcal{P}_d^{in}, \delta, \mathcal{P}_d^{out} \setminus \{p_{d'}\})$
3. *If d has an additional activation channel from another function node, then*
   - $\Sigma' = \Sigma \setminus \{w\}$, $\mathcal{P}' = \mathcal{P} \setminus \{p_d, p_w\}$, $\mathcal{D}' = \mathcal{D}$, $\mathcal{C}' = \mathcal{C} \setminus \{c_w\}$,
   - $f = (\mathcal{P}^{in}, (S, s_0, concat(T, p_a, p_w) \cup T_a), \mathcal{P}^{out} \setminus \{p_w\})$
     where $T_a = \{(p,\ E,\ s \rightarrow \Psi,\ s') \in T \mid p = p_a\}$
   - $d = (\mathcal{P}_d^{in} \setminus \{p_d\}, \delta, \mathcal{P}_d^{out})$ ◇



(a) Simple loop with local data node



(b) Complex loop with non-local data node

Fig. 5. Eliminating self-loops

The semantic consequence of this operation is mainly the change of causal event chains that include the events w and a. All these event chains are shortened by removing the sub-chain from w to a. This is realized by concatenating the appendant transitions. But even if these events are removed completely, the causality of the remaining observable events of the component is still preserved.

This is exemplified in Figure 5a where a function node with a self-activation is shown whose involved data node is local to that function node. The arrows in the function node indicate two transitions that are executed successively . On the right side the situation is shown after the ports 2 and 3 were removed by eliminating the self-activation. Here, the two transitions are concatenated. But an activation at port 1 still leads to an event at port 4 as on the left side. What is different, is the fact that both transitions are now executed as one transition. While on the left side it was possible that another activation occurs between these transitions it is not possible on the right side anymore. This reduces the set of possible execution traces. Figure 5b shows another example where the involved data node has a further outgoing activation channel to another function node f_2. Thus, the data node is still existent after self-loop elimination but the back-loop channel to f is removed. Additionally, the output port 2 still exists to activate f_2. So, even if the two transitions are concatenated to one transition, the firing of port 2 is maintained. This keeps the causality of the interface events to f_2. Concerning timing, the delay between any input and output event of the interface either stays the same (if it is not affected by the self-activation) or is even shortened because the delay of the self-activation (which is always > 0) is no longer existent. Furthermore, the number of task switches is reduced because two activations are now executed as one.

4.3 Task Creation Algorithm

The objective of the task creation algorithm is to partition function nodes into a set of at least m^- partitions while minimizing the cohesion function and

respecting the user-defined constraints. The function nodes of one partition are merged afterwards to one task by the previously defined operations. From the semantic point of view each two function nodes may be merged without violating any causality of events. But at least for Simulink models only nodes with the same period are allowed to be merged. The algorithm consists of two steps where first an initial solution is created by a constructive algorithm. Afterwards, an adapted state-of-the-art algorithm is applied to optimize the initial solution. Due to the fact that function networks derived from Simulink models typically consist of connected function nodes in several synchronous sets with a high amount of communication, the algorithm for the initial partitioning is communication-driven as well and works as follows:

1. Put each node $n \in N$ into an own separate partition.
2. For each channel c connecting two partitions T_i, T_j
   - Check if merging of $\{T_i, T_j\}$ is valid w.r.t to constraints,
   - Calculate gain G of cohesion by joining $\{T_i, T_j\}$.
3. Merge that pair of partitions $\{T_i, T_j\}$ with the maximum gain G.
4. Proceed until no valid set of merging candidates with $G > 0$ can be found.

The result is then improved by a combination of the Kernighan/Lin (KL) [10] and Fiduccia/Mattheyses (FM) [8] algorithms that move or exchange nodes between partitions. A discussion about complexity and optimality of these algorithms can be found in [3]. The complexity of the initial algorithm is $O(|N| \cdot |\mathcal{C}|)$.

The final result is a set of partitions of nodes while all function nodes of the same partition are merged to create a task. The order in which the nodes are merged is irrelevant, because the operation *merge* is associative. Empty partitions do not result in a task. To complete the task creation process, it is checked for each local data node and each self-activation if it can be eliminated with the appendant operation. Here, local data nodes have to be eliminated before self-activations are tackled, because local data nodes may induce read channels that prevent a semantic-preserving self-loop elimination. Even though both elimination operations are not absolutely necessary for task creation, they play an important role to reduce task switches and communication times.

5 Case Study and Experimental Results

As a case study to evaluate the presented approach we chose an advanced driver assistance system named *Virtual Driver Assistant (ViDAs)*[2] specified in Matlab Simulink as a single-rate model (i.e. one synchronous set). Beside a common adaptive cruise control it additionally contains a lane change assistant and a module to spot speed-limit signs to adjust the speed accordingly.

This model was translated automatically into a function network consisting of 140 nodes with 198 channels interconnecting them. We estimated worst case execution times for the translated blocks based on the code generated from Realtime Workshop Embedded Coder for a LEON3 processor running at 81 MHz, and annotated them as transition delays of the respective function nodes. To calculate

Table 1. Evaluation Results

System			Param.		Initial Val.		Result Values				Runtime
Name	#N	#C	m^-	$\overline{w}$	$\widehat{w}$	*com*	m	$min/max\ w(\tau)$	$\widehat{w}$	*com*	(seconds)
FN_1	50	70	5	0.261	0.235	0.511	6	0.169 / 0.267	0.054	0.058	4
FN_2	100	142	10	0.229	0.207	0.984	12	0.112 / 0.282	0.058	0.127	32
ViDAs	140	198	14	0.169	0.156	1.441	14	0.078 / 0.342	0.085	0.113	87
FN_3	200	284	20	0.235	0.212	1.897	23	0.141 / 0.290	0.053	0.116	158
FN_4	300	427	30	0.213	0.192	2.875	32	0.098 / 0.294	0.052	0.233	345

channel weights we assumed a FlexRay bus with a maximum bandwidth of 10 MBit/s. This function network was given as input to the proposed task creation algorithm. As user-defined parameters we set the minimum number of tasks m^- to 14 (aspired task weight $\overline{w}$=0.169), the maximum allowed task weight w^+ to 0.3 and α and β to 1. The initial node weights vary between 0.002 and 0.342 (average weight 0.017) with a standard deviation $\widehat{w}$ of 0.156 and a communication weight *com* of 1.441 (which would be infeasible for the bus). This is depicted in Table 1 in the first three big columns of line "ViDAs" where #N denotes the number of nodes and #C the number of channels. After a runtime of 87s (initial: 8s, KL/FM: 79s) we get a result where $\widehat{w}$ is reduced to 0.085 (factor 1.84) and *com* to 0.113 (factor 12.75). In Table 1 these results can be found in the columns "Result Values" and "Runtime". The reason why *com* could be reduced much more than $\widehat{w}$ is mainly that the initial system already has a comparably high communication weight leading to a greater potential for optimization of *com*. But still, the minimum task weight could be increased as desired from 0.002 to 0.078. Putting all nodes into one task as Realtime Workshop would do results in a task weight of 2.365 which would be infeasible.

To further evaluate the quality and scalability of the approach, we generated artificial function networks consisting of a number of function nodes with weights between 0.002 and 0.1. The input parameters restrict w^+ to 0.3 and m^- to 10% of #N. The corresponding results are shown in the remaining lines of Table 1. These results confirm the observations we made for the ViDAs system, because also here communication could be reduced significantly. The reason why the resulting $\widehat{w}$ is mostly lower as for ViDAs is due to the simplified communication structure of the generated networks and the assumption that initial node weights are distributed uniformly which is not always the case for real Simulink models.

6 Conclusions

The front-end process of a framework is presented for a design process that starts at high level specifications, and ends at implementations at the architecture level. Function networks are employed as an expressive interface between Simulink models forming the entry point of the process, and the following exploration process to find optimal hardware architectures executing the functions.

The task creation described in this paper forms the initial optimization step within the intended design process by obtaining a balanced task network that

minimizes task communication to avoid a typical bottleneck of bus utilization in distributed hardware architectures found in automotive industry. The paper does not only describe the algorithms of task creation but also provides a well-defined semantic foundation for composition of the function network formalism which is useful to reason for example about preserving semantics while merging nodes to larger task structures. Concerning future work it is planned to reduce the algorithms complexity and also evaluate alternative techniques. Additionally, the interaction with the succeeding design space exploration should be considered in more detail involving e.g. backtracking to learn from previous decisions.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126, 183–235 (1994)
2. Bao, J., Battram, P., Enkelmann, A., Gabel, A., Heyen, J., Koepke, T., Läsche, C., Sieverding, S.: Projektgruppe ViDAs - Endbericht (2010)
3. Büker, M., Damm, W., Ehmen, G., Metzner, A., Stierand, I., Thaden, E.: Automating the design flow for distributed embedded automotive applications: keeping your time promises, and optimizing costs, too. Technical Report 69, SFB/TR 14 AVACS (2011)
4. Büker, M., Metzner, A., Stierand, I.: Testing real-time task networks with functional extensions using model-checking. In: Proc. ETFA (2009)
5. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., Niebert, P.: From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In: Proc. ACM SIGPLAN, LCTES (2003)
6. Di Natale, M., Guo, L., Zeng, H., Sangiovanni-Vincentelli, A.: Synthesis of multitask implementations of simulink models with minimum delays. IEEE Transactions on Industrial Informatics (2010)
7. Ferdinand, C.: Worst-case execution time prediction by static program analysis. In: Proc. IPDPS (2004)
8. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proc. DAC 1982, pp. 175–181 (1982)
9. Jersak, M., Richter, K., Ernst, R.: Performance Analysis for Complex Embedded Applications. International Journal of Embedded Systems, Special Issue on Codesign for SoC (2004)
10. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal 49(1) (1970)
11. Kugele, S., Haberl, W.: Mapping data-flow dependencies onto distributed embedded systems. In: Proc. of SERP 2008 (2008)
12. Lublinerman, R., Tripakis, S.: Modular code generation from triggered and timed block diagrams. In: IEEE Real-Time and Embedded Technology and Applications Symposium, vol. 0, pp. 147–158 (2008)
13. Pouzet, M., Raymond, P.: Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In: Proc. of EMSOFT (2009)
14. Rox, J., Ernst, R.: Construction and Deconstruction of Hierarchical Event Streams with Multiple Hierarchical Layers. In: Proc. ECRTS (2008)

Performability Measure Specification: Combining CSRL and MSL

Alessandro Aldini[1], Marco Bernardo[1], and Jeremy Sproston[2]

[1] Università di Urbino "Carlo Bo" – Italy
[2] Università di Torino – Italy

Abstract. An integral part of the performance modeling process is the specification of the performability measures of interest. The notations proposed for this purpose can be grouped into classes that differ from each other in their expressiveness and usability. Two representative notations are the continuous stochastic reward logic CSRL and the measure specification language MSL. The former is a stochastic temporal logic formulating quantitative properties about states and paths, while the latter is a component-oriented specification language relying on a first-order logic for defining reward-based measures. In this paper, we combine CSRL and MSL in order to take advantage of the expressiveness of the former and the usability of the latter. To this aim, we develop a unified notation in which the core logic of MSL is employed to set up the reward structures needed in CSRL, whereas the measure definition mechanism of MSL is exploited to formalize measure and property specification patterns in a component-oriented fashion.

1 Introduction

The performance modeling process comprises two tasks: describing system evolution over time and specifying performability measures. The former task can be handled with a wide range of mature formalisms ranging from low level ones like continuous-time Markov chains (CTMC) and queueing networks to higher level and general-purpose ones like stochastic automata, stochastic Petri nets, and stochastic process calculi (see, e.g., [8,7] for a survey).

The latter task consists of expressing the metrics on the basis of which the system model should be assessed at steady state or at a given time instant or interval. The traditional way of doing this is to construct a reward structure [16] by associating real numbers with the states and the transitions of the CTMC underlying the system model. The reward assigned to a state determines the rate at which a gain/loss is accumulated while sojourning in that state, whereas the reward assigned to a transition determines the instantaneous gain/loss implied by the execution of that transition. The value of the measure is then computed as the sum of state probabilities and transitions frequencies, where rewards are used as weights.

In order to cope with the higher level of abstraction of general-purpose system modeling formalisms, more recent approaches to performability measure specification instead resort to a mixture of rewards and logics in a manner that avoids any direct intervention on the underlying CTMC (see, e.g., [4,3,1] and the references therein). In this paper, we focus on two representatives of the above mentioned approaches, which

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 165–179, 2011.

are CSRL and MSL, with the aim of combining their expressiveness and usability, thus enabling the modeler to take advantage of their complementary strengths.

CSRL [4,3] is a stochastic temporal logic suitable for expressing properties on system descriptions with an underlying CTMC semantics. These properties can be state-based, like, e.g., stating whether the steady-state probability of being in states satisfying a certain condition is in a specified relation with a given threshold, and path-based, like, e.g., stating whether the probability of observing paths satisfying a certain condition is in a specified relation with a given threshold.

MSL [1], which is composed of a core logic and a measure definition mechanism, has been conceived to support the specification of performability measures for descriptions of component-based systems with an underlying CTMC semantics. The core logic of MSL defines reward structures by assigning real numbers to states and transitions of the CTMC. Each state of the CTMC is viewed as a vector of local states representing the current behavior of the various components, and can be given a reward either directly on the basis of its constituent local states or indirectly on the basis of the activities labeling its outgoing transitions. The measure definition mechanism of MSL enhances usability by means of a component-oriented level on top of the core logic. The idea is to employ such a higher level to set up libraries of measure definitions that are provided by performability experts, which could then be exploited by nonexperts too.

Both MSL and CSRL suffer from some limitations. On the one hand, MSL is suitable for specifying classical measures such as throughput, utilization, queue length, and response time, but it is not adequate for formulating path properties. On the other hand, CSRL requires familiarity with temporal logic and supports neither the component-oriented formalization of performability measures nor the definition of policies for stating the association of rewards to states, thus complicating the work for nonexperts.

As a solution to such drawbacks, in this paper we propose UMSL, a unified measure specification language arising from the combination of two adequate extensions of MSL and CSRL. The objective is to enable the modeler to take advantage of the usability of the former and the expressiveness of the latter in a single notation. In order to make it affordable the specification of performability measures also by nonspecialists, the development of the unified language aims at a clear separation of concerns by means of (i) a core logic for setting up both reward structures and logical formulas and (ii) a measure definition mechanism for expressing performability measures on top of the core logic.

In the remainder of this paper, we introduce finite labeled CTMCs as a reference model for the assessment of the performability of component-based systems (Sect. 2), we present the core logic of UMSL (Sect. 3), we illustrate the measure definition mechanism of UMSL, which is an extension of that of MSL (Sect. 4), and finally we provide some concluding remarks (Sect. 5).

2 Reference Model

According to the guidelines of [2], the description of a component-based system should comprise at least the description of the individual system component types and the description of the overall system topology. The description of a single component type

should be provided by specifying at least its name, its parameters, its behavior, and its interactions. The overall behavior should express all the alternative sequences of activities that the component type can carry out – which can be formalized by means of traditional tools like automata, Petri nets, and process calculi – while the interactions are those activities used by the component type to communicate with the rest of the system.

For performability evaluation purposes, we assume that from the description of a component-based system it is possible to extract a stochastic model in the form of a CTMC. Since the overall behavior of each component can be observed through the activities that are executed, every transition of the CTMC is labeled not only with its rate $\lambda \in \mathbb{R}_{>0}$, but also with an action taken from a set Act. Likewise, every state of the CTMC is labeled with the vector of local states denoting the current behaviors of the $N \in \mathbb{N}_{>0}$ components in the system, with each local state belonging to a set Loc. In this way, the stochastic model reflects the component-oriented nature of the system description. Formally, our reference model is a finite labeled CTMC:

$$\mathcal{M} = (S, T, L, N, Loc, Act)$$

where S is a finite set of states, $T \subseteq S \times Act \times \mathbb{R}_{>0} \times S$ is a finitely-branching action-labeled transition relation, and $L : S \to Loc^N$ is an injective state-labeling function.

For notational convenience, $s \xrightarrow{a,\lambda}_{\mathcal{M}} s'$ denotes the transition $(s, a, \lambda, s') \in T$ and $L(s) = [z_1, z_2, \ldots, z_N]$ the vector of local states labeling $s \in S$. We use the notation $z \in s$ (resp. $z \notin s$) to express that $z = z_i$ for some $1 \leq i \leq N$ (resp. $z \neq z_i$ for all $1 \leq i \leq N$). Moreover, we denote with $negLoc$ (resp. $negAct$) the set of negations $\bar{z}$ (resp. $\bar{a}$) of local states $z \in Loc$ (resp. activities $a \in Act$). Negation is simply a shorthand for expressing that a component is not in a given local state or cannot execute a given activity. Finally, we define predicate sat by letting:

$$s \; sat \; z \text{ iff } z \in s \qquad s \; sat \; a \text{ iff } \exists \lambda \in \mathbb{R}_{>0}, s' \in S.\, s \xrightarrow{a,\lambda}_{\mathcal{M}} s'$$

$$s \; sat \; \bar{z} \text{ iff } z \notin s \qquad s \; sat \; \bar{a} \text{ iff } \nexists \lambda \in \mathbb{R}_{>0}, s' \in S.\, s \xrightarrow{a,\lambda}_{\mathcal{M}} s'$$

which is extended to conjunctions and disjunctions of local states and activities in the expected way. As an example, we have that $s \; sat \; z_1 \wedge z_2$ iff $s \; sat \; z_1$ and $s \; sat \; z_2$.

Throughout the paper, we use C as metavariable for components, B for specific current behaviors, a for activities, and the dot notation $C.B$ (resp. $C.a$) to express component behaviors (resp. component activities).

Example 1. Let us consider as a running example a system with two identical servers P_1 and P_2 that process requests arriving at the system with rate $\lambda \in \mathbb{R}_{>0}$. When a request finds both servers busy, it must immediately leave the system; i.e., no buffer is present. When a request finds both servers idle, it has the same probability to be accepted by the two servers. The two servers process incoming requests at rates $\mu_1, \mu_2 \in \mathbb{R}_{>0}$, respectively. They can also fail with rates $\chi_1, \chi_2 \in \mathbb{R}_{>0}$, respectively, and are then repaired with rates $\varrho_1, \varrho_2 \in \mathbb{R}_{>0}$, respectively.

Assuming that the arrival process has a single local state – $Arrivals$ – and that each server has three local states – $Idle$, $Busy$, and $Failed$ – the labeled CTMC underlying this system is as shown in Fig. 1, with the initial state being the leftmost one. For instance, the label $P_1.Idle$ in the initial state denotes that server P_1 is initially idle. ■

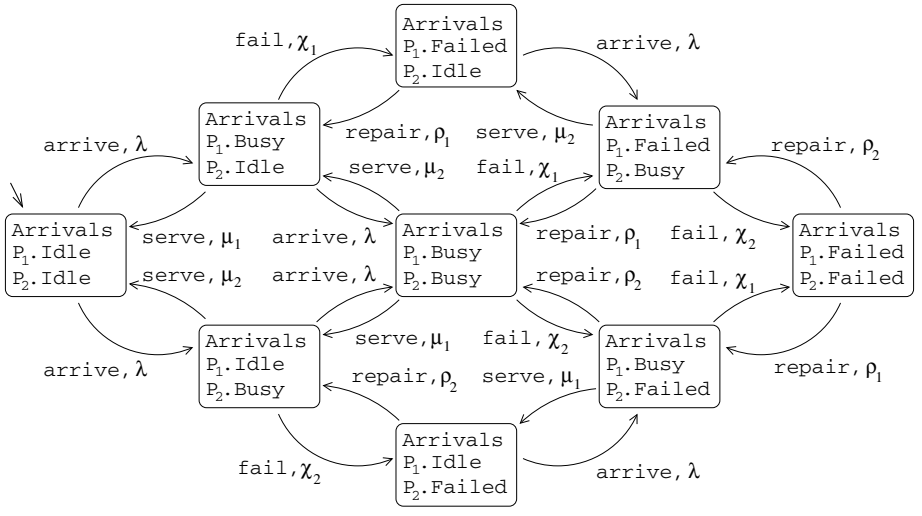


Fig. 1. Labeled CTMC for the running example

3 Core Logic of UMSL

In order to complement the expressiveness of CSRL with the capability of MSL of defining reward structures, in this section we combine the two formalisms into a single logic that will be the core of UMSL. First, we define an extension of MSL (Sect. 3.1) and an extension of CSRL (Sect. 3.2), then we show how they are combined (Sect. 3.3).

3.1 Extending MSL Parameterization

Each formula of the core logic of MSL is a first-order predicate parameterized with respect to a set of local states or activities that contribute to the value of a performability measure [1]. The role of the predicate is to specify how the contributions are combined to set the reward gained while sojourning in each state. In order to generalize the parameterization mechanism, we introduce an extension of MSL, called dnfMSL, in which each predicate is defined with respect to a set of groups of local states or activities expressed in disjunctive normal form for the sake of uniform treatment and usability.

The set *dnfLoc* of disjunctive normal forms on local states comprises elements like:

$$Z = (z_{1,1} \wedge \ldots \wedge z_{1,m_1}) \vee \ldots \vee (z_{n,1} \wedge \ldots \wedge z_{n,m_n})$$

which, to hide connectives to nonexperts, can be abbreviated with $Z = \{Z_1, \ldots, Z_n\}$, where $Z_i = \{z_{i,1}, \ldots, z_{i,m_i}\}$ and each literal $z_{i,j}$ occurring in conjunct Z_i is either a local state or the negation of a local state. In practice, conjunction expresses that a certain group of local states is needed for a state to gain a reward, while disjunction establishes that alternative conjuncts may contribute to assign a reward to the state.

Similarly, the set *dnfAct* of disjunctive normal forms on activities comprises elements such as:

$$A = (a_{1,1} \wedge \ldots \wedge a_{1,m_1}) \vee \ldots \vee (a_{n,1} \wedge \ldots \wedge a_{n,m_n})$$

which can be abbreviated with $A = \{A_1, \ldots, A_n\}$, where $A_i = \{a_{i,1}, \ldots, a_{i,m_i}\}$ and each literal $a_{i,j}$ occurring in A_i is either an activity or the negation of an activity.

We note that the use of conjunction constitutes a novelty with respect to the parameterization mechanism of [1] that enhances the expressive power. The case $m_i = 1$ for all $1 \leq i \leq n$, in which each conjunct is a singleton, corresponds to an original set parameter for MSL formulas of [1].

We now define four formula schemas that specify the way in which the contributions provided by each element in the set parameter are combined to establish a reward. These formulas arise from the combination of the type of elements forming the set parameter (local states or activities) with the fact that each (or only one) conjunct in the set parameter that is satisfied contributes to the reward. The predicates and functions that will be used subsequently are as follows:

- $eq : \mathbb{R} \times \mathbb{R} \to \{\text{true}, \text{false}\}$ such that:
$$eq(x, y) = \begin{cases} \text{true} & \text{if } x = y \\ \text{false} & \text{otherwise} \end{cases}$$
- $state\_rew : S \to \mathbb{R}$ such that $state\_rew(s)$ is the reward accumulated while sojourning in state s due to either local states of s or activities enabled by s.
- $lstate\_rew : Loc \cup negLoc \to \mathbb{R}$ such that $lstate\_rew(z)$ is the reward contribution given by local state z and $lstate\_rew(\bar{z})$ is the reward contribution given by the negation of local state $\bar{z}$.
- $act\_rew : Act \cup negAct \to \mathbb{R}$ such that $act\_rew(a)$ is the reward contribution given by activity a and $act\_rew(\bar{a})$ is the reward contribution given by the negation of activity $\bar{a}$.

The values $lstate\_rew(\_)$ and $act\_rew(\_)$ should be defined by the modeler depending on the measure of interest: in Sect. 4, we will show how to make such an assignment as transparent as possible. Then, each of the following formula schemas states that for a given set parameter the reward $state\_rew(\_)$ results from a specific combination of the contributions provided by local states or activities occurring in the set parameter.

The first formula schema assigns to $s \in S$ a direct state reward, to which all the conjuncts of local states in a set Z that are satisfied by s contribute in an additive way. The contribution of each conjunct $Z_i \in Z$ that is satisfied is obtained by combining the rewards associated with the literals occurring in Z_i through a function $af : 2^{\mathbb{R}} \to \mathbb{R}$ – like, e.g., sum, min, max, and avg – taken from a set AF of associative and commutative arithmetical functions. The first formula schema asserts the following for each $s \in S$:

$$\boxed{eq(state\_rew(s), sum\_lstate\_contrib(s, Z, af))}$$

where Z and af are given and $sum\_lstate\_contrib : S \times dnfLoc \times AF \to \mathbb{R}$ is such that:
$$sum\_lstate\_contrib(s, Z, af) = \sum_{Z_i \in Z \text{ s.t. } s \; sat \; Z_i} af\{\!|\, lstate\_rew(z) \mid z \in Z_i \,|\!\}$$
which is zero if there is no $Z_i \in Z$ satisfied by s.

The second formula schema assigns to $s \in S$ a direct state reward, to which only one among the conjuncts of local states in a set Z that are satisfied by s contributes. The contribution is selected among the contributions of those conjuncts by applying a function $cf : 2^{\mathbb{R}} \to \mathbb{R}$ – like, e.g., max and min – taken from a set CF of choice

functions; i.e., $cf(\emptyset) = 0$ and $cf(\{x_1, \ldots, x_n\}) \in \{x_1, \ldots, x_n\}$ for all $n \in \mathbb{N}_{>0}$. Formally, the second formula schema asserts the following for each $s \in S$:

$$\boxed{eq(state\_rew(s), choose\_lstate\_contrib(s, Z, af, cf))}$$

where cf is given too and $choose\_lstate\_contrib : S \times dnfLoc \times AF \times CF \to \mathbb{R}$ is such that:

$$choose\_lstate\_contrib(s, Z, af, cf) = \underset{Z_i \in Z \text{ s.t. } s \; sat \; Z_i}{cf} af\{\!|\, lstate\_rew(z) \mid z \in Z_i \,|\!\}$$

The third formula schema assigns to $s \in S$ an indirect state reward, to which all the conjuncts of activities in a set A that are satisfied by s contribute in an additive way. Formally, the third formula schema asserts the following for each $s \in S$:

$$\boxed{eq(state\_rew(s), sum\_act\_contrib(s, A, af))}$$

where A and af are given and $sum\_act\_contrib : S \times dnfAct \times AF \to \mathbb{R}$ is such that:

$$sum\_act\_contrib(s, A, af) = \sum_{A_i \in A \text{ s.t. } s \; sat \; A_i} af\{\!|\, act\_rew(a) \mid a \in A_i \,|\!\}$$

which is zero if there is no $A_i \in A$ satisfied by s.

The fourth formula schema assigns to $s \in S$ an indirect state reward, to which only one among the conjuncts of activities in a set A that are satisfied by s contributes. Formally, the fourth formula schema asserts the following for each $s \in S$:

$$\boxed{eq(state\_rew(s), choose\_act\_contrib(s, A, af, cf))}$$

where cf is given too and $choose\_act\_contrib : S \times dnfAct \times AF \times CF \to \mathbb{R}$ is such that:

$$choose\_act\_contrib(s, A, af, cf) = \underset{A_i \in A \text{ s.t. } s \; sat \; A_i}{cf} af\{\!|\, act\_rew(a) \mid a \in A_i \,|\!\}$$

Example 2. The system throughput for the running example can be determined by assigning to each state an indirect state reward obtained by summing up the rates of the *serve* activities enabled in that state, as shown by this formula of the third schema:

$$eq(state\_rew(s), sum\_act\_contrib(s, \{\{P_1.serve\}, \{P_2.serve\}\}, \text{sum}))$$

with $act\_rew(P_i.serve) = \mu_i$ for $i = 1, 2$. It is also possible to define the same measure by using direct state rewards as in the following formula of the first schema:

$$eq(state\_rew(s), sum\_lstate\_contrib(s, \{\{P_1.Busy\}, \{P_2.Busy\}\}, \text{sum}))$$

with $lstate\_rew(P_i.Busy) = \mu_i$ for $i = 1, 2$.

The system utilization can be specified by assigning to each state a unitary reward if the state includes at least one server in the $Busy$ local state, as specified by the following formula of the second schema:

$$eq(state\_rew(s), choose\_lstate\_contrib(s, \{\{P_1.Busy\}, \{P_2.Busy\}\}, \min, \min))$$

with $lstate\_rew(P_i.Busy) = 1$ for $i = 1, 2$.

As a variant of the previous measure, consider the utilization of server P_1 whenever server P_2 is not in the busy state, which can be defined as follows:

$$eq(state\_rew(s), choose\_lstate\_contrib(s, Z, \min, \min))$$

with $Z = \{\{P_1.Busy, P_2.Idle\}, \{P_1.Busy, P_2.Failed\}\}$, $lstate\_rew(P_1.Busy) = 1$, and $lstate\_rew(P_2.Failed) = lstate\_rew(P_2.Idle) = 1$. Alternatively, by using negation we have $Z = \{\{P_1.Busy, \overline{P_2.Busy}\}\}$ and $lstate\_rew(\overline{P_2.Busy}) = 1$. ■

We conclude by noting that, as done in [1], analogous formula schemas could be easily included in dnfMSL that assign rewards to transitions rather than states. However, they would not enhance the expressiveness of the logic.

3.2 Extending CSRL with Actions

The stochastic temporal logic CSRL expresses both state properties and path properties based on conditions over the states traversed along a path [4,3]. In an action-based, component-oriented setting, it is convenient to extend the temporal operators of CSRL with actions. The resulting action-based logic, which we call aCSRL, is actually a combination of CSRL and aCSL [14]. In contrast to aCSL, in aCSRL we allow reference to states. As in CSRL, the syntax of aCSRL features bounds on both the time and the reward accumulated along a path, while, as in aCSL, the syntax of aCSRL can make reference to actions exhibited along the path.

The syntax of the state formulas of aCSRL is as follows:

$$\boxed{\Phi \ ::= \ Z \mid A \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{S}_{\bowtie p}(\Phi) \mid \mathcal{P}_{\bowtie p}(\varphi) \mid \mathcal{E}_J(\Phi) \mid \mathcal{E}^t_J(\Phi) \mid \mathcal{C}^I_J(\Phi)}$$

where $Z \in \mathit{dnfLoc}$ is a disjunctive normal form on local states, $A \in \mathit{dnfAct}$ is a disjunctive normal form on activities, $\bowtie \in \{<, \leq, \geq, >\}$ is a comparison operator, $p \in \mathbb{R}_{[0,1]}$ is a probability, $t \in \mathbb{R}_{\geq 0}$ is a time value, I is an interval of time values, J is an interval of real-valued rewards, and φ is a path formula (see below).

State formulas are built from: disjunctive normal forms on local states or activities; logical conjunction and negation; the steady-state operator $\mathcal{S}_{\bowtie p}(\Phi)$, which establishes whether the steady-state probability of being in states that satisfy Φ is in relation $\bowtie$ with the threshold p; the probabilistic operator $\mathcal{P}_{\bowtie p}(\varphi)$, which establishes whether the probability of taking paths that satisfy φ is in relation $\bowtie$ with the threshold p; and the expected reward operators $\mathcal{E}_J(\Phi)$, $\mathcal{E}^t_J(\Phi)$, and $\mathcal{C}^I_J(\Phi)$ [4,3], which establish whether the reward accumulated at steady state or at a given time instant t or interval I while sojourning in states that satisfy a certain state formula Φ is in interval J.

Example 3. The formula $\mathcal{E}_{[0,7]}(P_1.\mathit{Failed})$ is true in a state if the expected reward accumulated in the long run in states in which $P_1.\mathit{Failed}$ holds is not greater than 7. The formula $\mathcal{E}^{10}_{[2,5]}(P_2.\mathit{Busy})$ is true in a state if the expected reward accumulated at time 10 in states in which $P_2.\mathit{Busy}$ holds is between 2 and 5. The formula $\mathcal{C}^{[0,15]}_{[8,\infty)}(\mathrm{true})$ is true in a state if the expected reward accumulated before 15 time units is at least 8. ■

The syntax of path formulas is as follows:

$$\boxed{\varphi \ ::= \ \Phi \ {}_{\mathcal{A}}U^{<t}_{<r} \ \Phi \mid \Phi \ {}_{\mathcal{A}_1}U^{<t}_{<r} \ {}_{\mathcal{A}_2} \ \Phi}$$

where $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2 \subseteq \mathit{Act}$ are sets of actions, $t \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a time value, and $r \in \mathbb{R} \cup \{\infty\}$ is a reward value.

The until formula $\Phi_1 \ {}_{\mathcal{A}}U^{<t}_{<r} \ \Phi_2$ is satisfied by an execution path if the path visits a state satisfying Φ_2 within t time units, while accumulating at most r reward, and visits states satisfying Φ_1 while performing only actions in $\mathcal{A}$ until that point. Similarly, the until formula $\Phi_1 \ {}_{\mathcal{A}_1}U^{<t}_{<r} \ {}_{\mathcal{A}_2} \ \Phi_2$ is satisfied by a path if the path visits a state satisfying Φ_2 within t time units, while accumulating at most r reward, after performing an action in $\mathcal{A}_2$, and visits states satisfying Φ_1 while performing only actions in $\mathcal{A}_1$ until that point. Note that a path satisfying $\Phi_1 \ {}_{\mathcal{A}_1}U^{<t}_{<r} \ {}_{\mathcal{A}_2} \ \Phi_2$ actually makes a transition to a state satisfying Φ_2, whereas this is not required in the case of $\Phi_1 \ {}_{\mathcal{A}}U^{<t}_{<r} \ \Phi_2$ (if the initial state of the path satisfies Φ_2).

Example 4. The formula $\mathcal{P}_{\leq 0.02}((P_1.Idle \vee P_1.Busy)\ {}_{\mathcal{A}}U^{\leq 30}_{<7}\ {}_{\{fail\}}\ P_1.Failed)$ is true in a state if, with probability at most 0.02, there is a point along the path at which the state property $P_1.Failed$ holds directly after performing any $fail$ action, at which no more than 30 time units have elapsed and at most 7 units of reward have been accumulated, and for which $P_1.Idle \vee P_1.Busy$ holds at all preceding points, and the actions in $\mathcal{A} = \{arrive, serve, repair\}$ are the only actions seen on the path before the action $fail$. Intuitively, the formula establishes whether certain conditions are met whenever along the path server P_1 fails for the first time (note that $P_1.Idle \vee P_1.Busy$ could be replaced by $\neg P_1.Failed$). ■

We restrict our attention to until formulas featuring upper bounds on time and accumulated rewards for simplicity (note that aCSL has been presented only with respect to upper bounds on time in [14], and that practical techniques for model-checking CSRL formulas feature upper bounds only in [3]).

Before presenting the semantics of aCSRL with respect to the reference model $\mathcal{M} = (S, T, L, N, Loc, Act)$, we introduce some notation. For each state $s \in S$, we let the exit rate of s be defined by $\mathrm{E}(s) = \sum\{\!|\ \lambda \in \mathbb{R}_{>0} \mid s \xrightarrow{a,\lambda}_{\mathcal{M}} s'\ |\!\}$. A state s is called absorbing if and only if $\mathrm{E}(s) = 0$. If $s \xrightarrow{a,\lambda}_{\mathcal{M}} s'$ and $t \in \mathbb{R}_{\geq 0}$, then we say that there exists a step of duration t from state s to state s' with action a, denoted by $s \xrightarrow{a,t} s'$. An infinite path is a sequence $s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} \cdots$ of steps. A finite path is a sequence $s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} \cdots \xrightarrow{a_{n-1},t_{n-1}} s_n$ of steps such that s_n is absorbing.

Let $Path^{\mathcal{M}}$ be the set of paths of $\mathcal{M}$ and let $Path^{\mathcal{M}}(s)$ be the subset of paths that commence in state s. For any state $s \in S$, let $Prob^{\mathcal{M}}_s$ denote the probability measure over the measurable subsets of $Path^{\mathcal{M}}(s)$ [5]. For any infinite path $\omega = s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} \cdots$ and any $i \in \mathbb{N}$, let $\omega(i) = s_i$, the $(i+1)$st state of ω, let $\delta(\omega, i) = t_i$, and, for $t \in \mathbb{R}_{\geq 0}$ and i the smallest index such that $t \leq \sum_{j=0}^{i} t_j$, let $\omega@t = \omega(i)$. For $\mathcal{A} \subseteq Act$, let $s_i \xrightarrow{\mathcal{A}} s_{i+1}$ be a predicate which is true if and only if $a_i \in \mathcal{A}$. For any finite path $\omega = s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} \cdots \xrightarrow{a_{l-1},t_{l-1}} s_l$, the state $\omega(i)$ and duration $\delta(\omega, i)$ are defined only if $i \leq l$, and are defined as in the infinite-path case (apart from $\delta(\omega, l)$, which equals ∞). Furthermore, for $t \geq \sum_{j=0}^{l-1} t_j$, let $\omega@t = \omega(l)$; otherwise, $\omega@t$ is defined as in the infinite-path case. Similarly, predicates of the form $s_i \xrightarrow{\mathcal{A}} s_{i+1}$ are defined only if $i < l$, and are defined as in the infinite-path case.

A transient probability is the probability of being in a certain state s' at time t given an initial state s. In the model-checking context, we can express a transient probability in terms of paths as $\pi^{\mathcal{M}}(s, s', t) = Prob^{\mathcal{M}}_s\{\omega \in Path^{\mathcal{M}}(s) \mid \omega@t = s'\}$. The steady-state probabilities are used to refer to the long-run average probability of the CTMC being in a state, and are defined by $\pi^{\mathcal{M}}(s, s') = \lim_{t\to\infty} \pi^{\mathcal{M}}(s, s', t)$. For $S' \subseteq S$, let $\pi^{\mathcal{M}}(s, S', t) = \sum_{s' \in S'} \pi^{\mathcal{M}}(s, s', t)$ and $\pi^{\mathcal{M}}(s, S') = \sum_{s' \in S'} \pi^{\mathcal{M}}(s, s')$.

Given a reward structure $\rho : S \to \mathbb{R}$, let the instantaneous reward $\rho^{\mathcal{M}}(s, s', t) = \pi^{\mathcal{M}}(s, s', t) \cdot \rho(s')$ and the expected long-run reward $\rho^{\mathcal{M}}(s, s') = \pi^{\mathcal{M}}(s, s') \cdot \rho(s')$. For $S' \subseteq S$, let $\rho^{\mathcal{M}}(s, S', t) = \sum_{s' \in S'} \rho^{\mathcal{M}}(s, s', t)$ and $\rho^{\mathcal{M}}(s, S') = \sum_{s' \in S'} \rho^{\mathcal{M}}(s, s')$. For an infinite path $\omega = s_0 \xrightarrow{a_0,t_0} s_1 \xrightarrow{a_1,t_1} \cdots$ and $i \in \mathbb{N}$, let $\gamma(\omega, i) = t_i \cdot \rho(s_i)$.

Table 1. Semantics of aCSRL

$s \models_{\mathcal{M}} Z$	iff $s \; sat \; Z$
$s \models_{\mathcal{M}} A$	iff $s \; sat \; A$
$s \models_{\mathcal{M}} \Phi_1 \wedge \Phi_2$	iff $s \models_{\mathcal{M}} \Phi_1$ and $s \models_{\mathcal{M}} \Phi_2$
$s \models_{\mathcal{M}} \neg\Phi$	iff $s \not\models_{\mathcal{M}} \Phi$
$s \models_{\mathcal{M}} \mathcal{S}_{\bowtie p}(\Phi)$	iff $\pi^{\mathcal{M}}(s, Sat^{\mathcal{M}}(\Phi)) \bowtie p$
$s \models_{\mathcal{M}} \mathcal{P}_{\bowtie p}(\varphi)$	iff $Prob_s^{\mathcal{M}}\{\omega \in Path^{\mathcal{M}} \mid \omega \models_{\mathcal{M}} \varphi\} \bowtie p$
$s \models_{\mathcal{M}} \mathcal{E}_J(\Phi)$	iff $\rho^{\mathcal{M}}(s, Sat^{\mathcal{M}}(\Phi)) \in J$
$s \models_{\mathcal{M}} \mathcal{E}_J^t(\Phi)$	iff $\rho^{\mathcal{M}}(s, Sat^{\mathcal{M}}(\Phi), t) \in J$
$s \models_{\mathcal{M}} \mathcal{C}_J^I(\Phi)$	iff $\int_I \rho^{\mathcal{M}}(s, Sat^{\mathcal{M}}(\Phi), u)\,du \in J$
$\omega \models_{\mathcal{M}} \Phi_1 \, {}_{\mathcal{A}}U_{<r}^{\leq t} \, \Phi_2$	iff $\exists k \geq 0.$ $(\omega(k) \models_{\mathcal{M}} \Phi_2 \wedge$ $(\forall i < k.\, \omega(i) \models_{\mathcal{M}} \Phi_1 \wedge \omega(i) \xrightarrow{\mathcal{A}} \omega(i+1)) \wedge$ $t \geq \sum_{i=0}^{k-1} \delta(\omega, i) \wedge r \geq \sum_{i=0}^{k-1} \gamma(\omega, i))$
$\omega \models_{\mathcal{M}} \Phi_1 \, {}_{\mathcal{A}_1}U_{<r}^{<t} \, {}_{\mathcal{A}_2} \, \Phi_2$	iff $\exists k > 0.$ $(\omega(k) \models_{\mathcal{M}} \Phi_2 \wedge$ $(\forall i < k-1.\, \omega(i) \models_{\mathcal{M}} \Phi_1 \wedge \omega(i) \xrightarrow{\mathcal{A}_1} \omega(i+1)) \wedge$ $\omega(k-1) \models_{\mathcal{M}} \Phi_1 \wedge \omega(k-1) \xrightarrow{\mathcal{A}_2} \omega(k) \wedge$ $t \geq \sum_{i=0}^{k-1} \delta(\omega, i) \wedge r \geq \sum_{i=0}^{k-1} \gamma(\omega, i))$

where $Sat^{\mathcal{M}}(\Phi) = \{s \in S \mid s \models_{\mathcal{M}} \Phi\}$.

After enriching the reference model $\mathcal{M}$ with a reward structure ρ on states only, the formal semantics of aCSRL is given by the satisfaction relation $\models_{\mathcal{M}}$ defined in Table 1.

3.3 Intertwining aCSRL and dnfMSL

The objective of UMSL is to combine aCSRL and dnfMSL in order to join their complementary advantages. In fact, on the one hand, dnfMSL is not expressive enough to establish that a state is given a reward only if it satisfies a complex condition formalized by a temporal logic formula that includes not only logical connectives. On the other hand, aCSRL is intended to specify logical properties but it does not help the modeler to understand which rewards must be attached to every state for any occurrence of until and expected reward operators.

In order to overcome these drawbacks, we propose two different ways of combining aCSRL formulas and dnfMSL formula schemas, which result in two intertwined notations, called dnfMSL+ and aCSRL+, that constitute the core of UMSL. Formally, a formula of UMSL can be a dnfMSL+ formula schema ν or an aCSRL+ formula Φ, such that:

$$
\begin{aligned}
\nu ::= \; & eq(state\_rew(s), sum\_lstate\_contrib_{\Phi}(s, Z, af)) \\
\mid \; & eq(state\_rew(s), choose\_lstate\_contrib_{\Phi}(s, Z, af, cf)) \\
\mid \; & eq(state\_rew(s), sum\_act\_contrib_{\Phi}(s, A, af)) \\
\mid \; & eq(state\_rew(s), choose\_act\_contrib_{\Phi}(s, A, af, cf))
\end{aligned}
$$

where:

$$sum\_lstate\_contrib_{\Phi}(s, Z, af) = \begin{cases} sum\_lstate\_contrib(s, Z, af) & \text{if } s \models \Phi \\ 0 & \text{otherwise} \end{cases}$$

and with the other three functions that are defined similarly, while:

$$\boxed{\Phi ::= Z \mid A \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{S}_{\bowtie p}(\Phi) \mid \mathcal{P}_{\bowtie p}(\nu, \varphi) \mid \mathcal{E}_J(\nu, \Phi) \mid \mathcal{E}^t_J(\nu, \Phi) \mid \mathcal{C}^I_J(\nu, \Phi)}$$

where φ is a path formula and ν provides probabilistic and reward operators with the reward structures that are needed for their interpretation.

A formula of dnfMSL+ parameterized with respect to Φ associates with each state $s \in S$ satisfying Φ the reward defined by the underlying dnfMSL formula schema. Note that dnfMSL+ extends dnfMSL in a conservative way. Indeed, the dnfMSL+ formula schema $eq(state\_rew(s), sum\_lstate\_contrib_{\Phi}(s, Z, af))$ and the dnfMSL formula schema $eq(state\_rew(s), sum\_lstate\_contrib(s, Z, af))$ define the same reward structure whenever either $\Phi = \mathrm{true}$ or $\Phi = Z$.

A formula of aCSRL+ is an extension of a formula of aCSRL in which every subformula requiring a reward structure ρ for its interpretation is paired with a dnfMSL+ formula schema ν defining such a reward structure. The semantics of aCSRL+ is exactly as shown in Table 1 in the case of aCSRL, with the assumption that regarding the reward structure ρ we have that $\forall s \in S.\, \rho(s) = state\_rew(s)$, where $state\_rew(s)$ is the reward assigned to s by ν.

Example 5. The formula $\mathcal{E}_{[0.5,0.7]}(\nu, P_1.Busy \vee P_2.Busy)$, where ν is the dnfMSL+ formula corresponding to the third case of Ex. 2, is true in a state if the system utilization, in the long run, is in the interval $[0.5, 0.7]$. ■

Thanks to the way in which dnfMSL+ and aCSRL+ are defined, the core logic of UMSL allows for a controlled form of nesting according to which a formula schema of dnfMSL+ embeds a formula of aCSRL+, while in turn a formula of aCSRL+ may embed a formula schema of dnfMSL+. In this way, UMSL offers the same expressiveness as aCSRL, whose operators are fully integrated in UMSL. Moreover, as we will see in the next section, UMSL includes additional capabilities concerned with the use of nested reward structures.

4 Measure Definition Mechanism of UMSL

MSL is equipped with a component-oriented measure definition mechanism built on top of the core logic for enhancing its usability [1]. In this section, we show that the same mechanism can be applied on top of dnfMSL+. Moreover, we show that it can be extended to aCSRL+ formulas so as to develop a component-oriented property definition mechanism. The combination of the two mechanisms makes the specification of UMSL formulas an easier task with respect to using CSRL, especially for people not familiar with (temporal) logics. The syntax for specifying in MSL a performability measure as a macro definition possibly parameterized with respect to a set of component-oriented arguments is as follows:

$$\texttt{MEASURE} \triangleleft name \triangleright \, (\triangleleft arguments \triangleright) \, \texttt{IS} \triangleleft body \triangleright$$

The metric is given a symbolic name and is parameterized with respect to component behaviors and component activities to be used in its body, which is defined in terms of MSL core logic formulas in the case of a basic measure. Assuming that the identifier of a metric denotes the value of the metric computed on a certain finite labeled CTMC, it is then possible to define derived measures whose body comprises metric identifiers combined through the usual arithmetical operators and mathematical functions. The idea is that libraries of basic measure definitions should be provided by performability experts, which could then be exploited by nonexperts too upon defining derived measures. In any case, when the definition of a measure is available, the modeler is only asked to provide component-oriented parameters without having to consider which numbers have to be associated with which entities.

We now show how this mechanism is inherited by dnfMSL+. The syntax for defining a performability measure in UMSL is extended as follows:

$$\boxed{\texttt{MEASURE} \lhd \mathit{name} \rhd (\lhd \mathit{arg1}; \mathit{arg2}; \mathit{arg3} \rhd) \texttt{ IS } \lhd \mathit{body} \rhd}$$

where the body of a basic measure is a dnfMSL+ formula schema ν, while the arguments are divided into three parts:

- $\mathit{arg1} ::= Z \mid A$
- $\mathit{arg2} ::= \Phi$
- $\mathit{arg3} ::= \infty \mid t \mid [t_1, t_2]$

The first two arguments represent the sequence of lists of component behaviors (or activities) parameterizing ν and the aCSRL+ formula embedded in ν, respectively. The third argument does not define the reward structure underlying the measure description. Instead, it is used for analysis purposes to specify the time at which the measure should be computed: ∞ denotes steady-state analysis, $t \in \mathbb{R}_{\geq 0}$ denotes instant-of-time analysis, and $[t_1, t_2]$ (with $t_1, t_2 \in \mathbb{R}_{\geq 0}$ and $t_1 < t_2$) denotes interval-of-time analysis.

The definition of Φ may be hard from the viewpoint of a modeler who is not familiar with temporal logics like aCSRL. For this reason, we now introduce a property definition mechanism on top of aCSRL+, which is inspired by the measure definition mechanism of MSL. The syntax for defining a property in UMSL is as follows:

$$\boxed{\texttt{PROPERTY} \lhd \mathit{name} \rhd (\lhd \mathit{arguments} \rhd) \texttt{ IS } \lhd \mathit{body} \rhd}$$

where the body is defined as an aCSRL+ formula parameterized with respect to the arguments that are provided. Arguments are given in form of a list ℓ defined as follows:

$$\ell ::= \ell' \mid \ell', \ell$$
$$\ell' ::= Z \mid A \mid \bowtie p \mid t \mid r \mid I \mid J$$

where Z (resp. A) is a sequence of lists of component behaviors (resp. activities) forming disjunctive normal forms, p is a probability and $\bowtie$ is a comparison operator used in steady-state and probabilistic operators, t (resp. r) is a time (resp. reward) value used in path formulas, while I (resp. J) is an interval used to specify time (resp. reward) bounds in expected reward operators.

In order to illustrate the measure definition mechanism of UMSL, let us consider three basic definitions, which will be used in the following examples.

The property determining whether a state satisfies a disjunctive normal form on local states $\{Z_1, \ldots, Z_n\}$, where $Z_i = \{z_{i,1}, \ldots, z_{i,m_i}\}$ for all $1 \leq i \leq n$, can be represented by the following definition:

$$\texttt{PROPERTY}\ sat\_elem(Z_1, ..., Z_n)\ \texttt{IS}\ (z_{1,1} \wedge ... \wedge z_{1,m_1}) \vee ... \vee (z_{n,1} \wedge ... \wedge z_{n,m_n})$$

The property that states whether the steady-state probability of being in a certain combination $Z \in dnfLoc$ of component behaviors is less than p is given by:

$$\texttt{PROPERTY}\ ss\_beh(Z, <p)\ \texttt{IS}\ \mathcal{S}_{<p}(Z)$$

The property establishing whether the probability of being for the first time in the component behavior $C.B$ by time t after having consumed at most an amount r of resources, with each unitary resource usage expressed by the execution of $C'.a$, is less than p, can be defined as:

$$\texttt{PROPERTY}\ path\_beh(C'.a, C.B, t, r, <p)\ \texttt{IS}\ \mathcal{P}_{<p}(\nu, \neg C.B\ {}_{Act}U^{\leq t}_{<r}\ C.B)$$

where ν is the formula $eq(state\_rew(s), sum\_act\_contrib_{\text{true}}(s, C'.a, \text{sum}))$ such that $act\_rew(C'.a) = 1$. This property generalizes Ex. 4 and emphasizes the support provided by dnfMSL+ to the definition of the reward structure needed by the interpretation of an until formula of aCSRL+.

Similar to the measure definition mechanism, assuming that the identifier of a property denotes the truth value of the corresponding aCSRL+ formula computed on a certain finite labeled CTMC, it is then possible to define derived properties whose body comprises property identifiers combined through the usual logical operators.

A property identifier can in turn be used as a macro in the definition of performability measures, in a way that masks the temporal logic formula that constitutes the body of the property. We illustrate this fact by considering two typical performance measures: system throughput and resource utilization.

As observed in Sect. 3.1, the definition of the throughput is parameterized with respect to the component activities $C_1.a_1, \ldots, C_n.a_n$ that contribute to the throughput. We refine this definition by assuming that a side condition – expressed by an aCSRL+ formula Φ – can be introduced to represent a guard that must be satisfied to count the activity contribution. Then the rate at which each state satisfying Φ accumulates reward is the sum of the rates of the contributing activities that are enabled at that state. If Φ is given a property definition $prop$ included in a library of properties, then we have the following measure definition:

$$\texttt{MEASURE}\ throughput(C_1.a_1, \ldots, C_n.a_n; prop(\_); \_)\ \texttt{IS}$$
$$eq(state\_rew(s), sum\_act\_contrib_{prop(\_)}(s, A, \text{sum}))$$

where $A = \{\{C_1.a_1\}, \ldots, \{C_n.a_n\}\}$ and $act\_rew(C_i.a_i) = \lambda_i$ for all $1 \leq i \leq n$ if the rate associated with $C_i.a_i$ is λ_i.

Example 6. Given the basic measure definition $throughput$, the average system throughput for the running example can be determined through the following invocation:

$$throughput(P_1.serve, P_2.serve; \text{true}; \infty)$$

which specifies the reward structure with $act\_rew(P_i.serve) = \mu_i$ for $i = 1, 2$. If the contribution of a server activity must be counted only when the other server is under repair, then the invocation becomes:

$$throughput(P_1.serve, P_2.serve;\ sat\_elem(\{P_1.Busy, P_2.Failed\}, \{P_2.Busy, P_1.Failed\});\ \infty)$$

■

In the case of the utilization of a resource, as seen in Sect. 3.1 we have to specify the set of component activities $C_1.a_1, \ldots, C_n.a_n$ modeling the utilization of that resource, while a unit reward is transparently associated with each state in which this activity is enabled. As in the case of the system throughput, we enrich the definition by adding a side condition:

$$\texttt{MEASURE}\ \mathit{utilization}(C_1.a_1, \ldots, C_n.a_n; \mathit{prop}(\_); \_)\ \texttt{IS}$$
$$\mathit{eq}(\mathit{state\_rew}(s), \mathit{choose\_act\_contrib}_{\mathit{prop}(\_)}(s, A, \mathrm{sum}, \min))$$

where $A = \{\{C_1.a_1\}, \ldots, \{C_n.a_n\}\}$ and $\mathit{act\_rew}(C_i.a_i) = 1$ for all $1 \leq i \leq n$.

Example 7. The utilization of server P_1 by time t is specified as follows:

$$\mathit{utilization}(P_1.\mathit{serve}; \mathrm{true}; [0, t])$$

Assume to be interested in counting the use of P_1 only if the probability of observing a subsequent P_1 failure by time t' after having been used at most r times is less than p. In this case, the invocation of the basic measure definition $\mathit{utilization}$ becomes:

$$\mathit{utilization}(P_1.\mathit{serve}; \mathit{path\_beh}(P_1.\mathit{serve}, P_1.\mathit{Failed}, t', r, <p); [0, t])$$ ■

An interesting set of measure definitions refers to the problem of determining the probability of being in specific component behaviors. In this case, it should be enough for the modeler to specify these component behaviors in terms of $Z \in \mathit{dnfLoc}$:

$$\texttt{MEASURE}\ \mathit{beh\_prob}(Z; Z; \_)\ \texttt{IS}$$
$$\mathit{eq}(\mathit{state\_rew}(s), \mathit{choose\_lstate\_contrib}_Z(s, Z, \min, \min))$$

such that $\mathit{lstate\_rew}(z) = 1$ when z is one of the component behaviors occurring in Z.

Example 8. The probability on the long run of being in a state in which both servers are under repair or both servers are idle is determined by taking:

$$Z = \{\{P_1.\mathit{Failed}, P_2.\mathit{Failed}\}, \{P_1.\mathit{Idle}, P_2.\mathit{Idle}\}\}$$

and then by using the invocation $\mathit{beh\_prob}(Z; Z; \infty)$. ■

The measure $\mathit{beh\_prob}$ can be generalized to express more complex measures like:

$$\texttt{MEASURE}\ \mathit{beh\_prob}(Z; \mathit{prop}(\_); \_)\ \texttt{IS}$$
$$\mathit{eq}(\mathit{state\_rew}(s), \mathit{choose\_lstate\_contrib}_{\mathit{prop}(\_)}(s, Z, \min, \min))$$

The measure above quantifies the probability of being in states satisfying the property $\mathit{prop}(\_)$ and including a combination of component behaviors occurring in Z.

Example 9. Consider the probability of being in the component behavior $P_1.\mathit{Failed}$ at time t, provided that with probability less than p we observe again action $P_1.\mathit{fail}$ by n time units while accumulating a number of arrivals less than r. The value is given by:

$$\mathit{beh\_prob}(P_1.\mathit{Failed}; \mathit{path\_beh}(\mathit{Arrivals}.\mathit{arrive}, P_1.\mathit{fail}, n, r, <p); t)$$

Now, consider an extension in which a failure state $P_1.\mathit{Failed}$ is taken into account also in the case that the related steady-state probability of being in a total failure state – i.e., both servers are under repair – is less than q. In this case, the invocation becomes:

$$\begin{array}{l}\mathit{beh\_prob}(P_1.\mathit{Failed};\\ \quad \mathit{path\_beh}(\mathit{Arrivals}.\mathit{arrive}, P_1.\mathit{fail}, n, r, <p) \vee\\ \quad\ \mathit{ss\_beh}(\{\{P_1.\mathit{Failed}, P_2.\mathit{Failed}\}\}, <q);\\ \quad t)\end{array}$$ ■

Finally, the parameterization of the measure definition mechanism of dnfMSL+ can be further extended by assuming that each literal occurring in Z (resp. A) is possibly associated with a real number expressing the reward contribution of the local state (resp. activity) to be used in the definition of function $\mathit{lstate\_rew}$ (resp. $\mathit{act\_rew}$).

For instance, the overall energy consumption with respect to the component behaviors $C.B_1, \ldots, C.B_n$ is the sum of the probabilities of being in the various local states, each multiplied by a reward that describes the rate at which energy is consumed in that state. Hence, we extend the basic measure beh_prob to derive the following measure:

$\texttt{MEASURE}\ energy\_consumption(C.B_1(l_1), C.B_2(l_2), \ldots, C.B_n(l_n); \text{true}; \_)\ \texttt{IS}$
$beh\_prob(C.B_1(l_1); \text{true}; \_) + \ldots + beh\_prob(C.B_n(l_n); \text{true}; \_)$

where $lstate\_rew(z) = l_i$ when $z = C.B_i$ for some $1 \leq i \leq n$.

Example 10. Assuming that the energy consumed in the busy state is 50% more than the energy consumed in the failure state, while no energy is consumed in the idle state, the overall energy consumption with respect to server P_1 in the interval $[0, t]$ is given by the following invocation of the derived measure $energy\_consumption$:

$$energy\_consumption(P_1.Idle(0), P_1.Busy(3), P_1.Failed(2); \text{true}; [0, t]).$$ ■

5 Conclusion

In this paper, we have shown how to combine two orthogonal extensions of the logics CSRL and MSL in order to obtain UMSL, a unified measure specification language trading expressiveness and usability.

From the expressiveness standpoint, the core logic of UMSL – i.e., the combination of dnfMSL+ and aCSRL+ – offers interesting features.

On the one hand, the expressiveness gain with respect to MSL core logic is twofold. Firstly, the combinations of local states and activities that contribute to the reward structure are now formalized as more expressive propositional logic formulas in disjunctive normal form. Secondly, it is possible to add conditions stating that certain states are given certain rewards only if they satisfy temporal logic formulas expressed in aCSRL+.

On the other hand, observing that aCSRL+ inherits the same expressiveness as CSL-like stochastic logics [14,3,18], several examples of Sect. 4 – see Exs. 7 and 9 – show that UMSL supports the definition of nested, independent reward structures. This feature is not considered in other mechanisms proposed in the literature for the definition of reward structures in the setting of stochastic model checking [10,19,9,17,18].

To complete the expressiveness analysis, it would be interesting to compare UMSL with alternative mechanisms for the specification of performance measures, like, e.g., Performance Trees, which have been proved to be more expressive than CSL [20].

Finally, from the usability standpoint, we observe that the objective of the definition mechanisms presented in Sect. 4 is to manage both reward structures and logic operators as transparently as possible, especially through the macro mechanism used for derived definitions. In this way, while basic definitions represent a task for experts, their use within derived definitions should be affordable by non-specialists. Along this line of research, it would be interesting to employ the measure and property definition mechanisms of UMSL to define specification pattern systems inspired by, e.g., ProProST [13]. Indeed, since measure and property definition mechanisms of UMSL can be combined to realize nested patterns in a component-oriented fashion, we believe that UMSL-based specification patterns would help practitioners to apply in a correct and easy way model-checking techniques for the verification of the performability of component-based systems.

Acknowledgment. This work has been funded by MIUR-PRIN project *PaCo – Performability-Aware Computing: Logics, Models, and Languages*.

References

1. Aldini, A., Bernardo, M.: Mixing Logics and Rewards for the Component-Oriented Specification of Performance Measures. Theoretical Computer Sc. 382, 3–23 (2007)
2. Aldini, A., Bernardo, M., Corradini, F.: A Process Algebraic Approach to Software Architecture Design. Springer, Heidelberg (2010)
3. Baier, C., Cloth, L., Haverkort, B., Hermanns, H., Katoen, J.-P.: Performability Assessment by Model Checking of Markov Reward Models. Formal Methods in System Design 36, 1–36 (2010)
4. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: On the Logical Characterisation of Performability Properties. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 780–792. Springer, Heidelberg (2000)
5. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-Checking Algorithms for Continuous-Time Markov Chains. IEEE Trans. on Software Eng. 29, 524–541 (2003)
6. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
7. Bernardo, M., Hillston, J. (eds.): Formal Methods for Performance Evaluation. LNCS, vol. 4486. Springer, Heidelberg (2007)
8. Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.): Lectures on Formal Methods and Performance Analysis. LNCS, vol. 2090. Springer, Heidelberg (2001)
9. Courtney, T., Daly, D., Derisavi, S., Gaonkar, S., Griffith, M., Lam, V., Sanders, W.: The Möbius Modeling Environment: Recent Developments. In: Proc. of the 1st Int. Conf. on Quantitative Evaluation of Systems (QEST 2004), pp. 328–329. IEEE-CS Press, Los Alamitos (2004)
10. Clark, G., Hillston, J.: Towards Automatic Derivation of Performance Measures from PEPA Models. In: Proc. of the 12th UK Performance Engineering Workshop (1996)
11. De Nicola, R., Katoen, J.-P., Latella, D., Loreti, M., Massink, M.: Model Checking Mobile Stochastic Logic. Theoretical Computer Science 382, 42–70 (2007)
12. De Nicola, R., Vaandrager, F.: Action Versus State Based Logics for Transition Systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
13. Grunske, L.: Specification Patterns for Probabilistic Quality Properties. In: Proc. of the 30th Int. Conf. on Software Engineering (ICSE 2008), pp. 31–40. ACM Press, New York (2008)
14. Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., Siegle, M.: Towards model checking stochastic process algebra. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, p. 420. Springer, Heidelberg (2000)
15. Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., Siegle, M.: Model Checking Stochastic Process Algebra, Technical Rep. IMMD7-2/00, University of Erlangen-Nürnberg (2000)
16. Howard, R.A.: Dynamic Probabilistic Systems. John Wiley & Sons, Chichester (1971)
17. Katoen, J.-P., Khattri, M., Zapreev, I.S.: A Markov Reward Model Checker. In: Proc. of the 2nd Int. Conf. on Quantitative Evaluation of Systems (QEST 2005), pp. 243–244. IEEE-CS Press, Los Alamitos (2005)
18. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
19. Obal, W., Sanders, W.: State-Space Support for Path-Based Reward Variables. Performance Evaluation 35, 233–251 (1999)
20. Suto, T., Bradley, J.T., Knottenbelt, W.J.: Performance Trees: Expressiveness and Quantitative Semantics. In: Proc. of the 4th Int. Conf. on the Quantitative Evaluation of Systems (QEST 2007), pp. 41–50. IEEE-CS Press, Los Alamitos (2007)

Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit Using CADP*

Etienne Lantreibecq[1] and Wendelin Serwe[2]

[1] STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France
Etienne.Lantreibecq@st.com
[2] INRIA/LIG, 655, av. de l'Europe, Inovallée, Montbonnot, 38334 St Ismier, France
Wendelin.Serwe@inria.fr

Abstract. The complexity of multiprocessor architectures for mobile multi-media applications renders their validation challenging. In addition, to provide the necessary flexibility, a part of the functionality is realized by software. Thus, a formal model has to take into account both hardware and software. In this paper we report on the use of LOTOS NT and CADP for the formal modeling and analysis of the DTD (Dynamic Task Dispatcher), a complex hardware block of an industrial hardware architecture developed by STMicroelectronics. Using LOTOS NT facilitated exploration of alternative design choices and increased the confidence in the DTD, by, on the one hand, automatic analysis of formal models easily understood by the architect of the DTD, and, on the other hand, co-simulation of the formal model with the implementation used for synthesis.

1 Introduction

Multi-media applications require complex multiprocessor architectures, even for mobile terminals such as smartphones or netbooks. Due to physical constraints, in particular the distribution of a global clock on large circuits, modern multiprocessor architectures for mobile multi-media applications are implemented using a globally asynchronous, locally synchronous (GALS) approach, combining a set of synchronous blocks using an asynchronous communication scheme.

Due to the high cost of chip-fabrication, errors in the architecture have to be found as early as possible. Therefore, architects are interested in applying formal methods in the design phase. In addition, a formal model has to take into account both hardware and software, because a part of the system's functionality is implemented in software to provide the flexibility required by the rapidly evolving market. However, even if the software part can be updated easily, the basic functionalities implemented in hardware have to be thoroughly verified.

This paper reports on the application of a modern formal analysis tool (CADP 2010 [4]), and in particular the LOTOS NT [3,7] language, to a complex

* This work has been partly funded by the French Ministry of Economics and Industry and by the *Conseil Général de l'Isère* (Minalogic project Multival).

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 180–195, 2011.

hardware block of an industrial architecture developed by STMicroelectronics, namely the Dynamic Task Dispatcher (DTD). The DTD serves to dispatch data-intensive applications on a cluster of processors for parallel execution.

Until to now, formal methods have been used by STMicroelectronics mainly for checking the equivalence between different steps in the design flow (e.g. between a netlist and a placed and routed netlist) or for establishing the correctness of a computational block (e.g. an inverse discrete cosine transform) by theorem proving. However, STMicroelectronics is unfamiliar with formal methods to validate a control block such as the DTD. For this reason, STMicroelectronics participates in research projects, such as the Multival[1] project on the validation of multiprocessor architectures using CADP. Our choice of CADP was also motivated by related successful case-studies, in particular the analysis of a system of synchronous automata communicating asynchronously [8], and the co-simulation of complex hardware circuits for cache-coherency protocols with their formal models [9]. Finally, because the considered design is a GALS architecture, the interfaces between the processors and the DTD can be considered asynchronous, which fits well with the modeling style supported by CADP.

We show several advantages of modeling and analyzing the DTD using LOTOS NT, a new formal language based on process algebra and functional programming, instead of classical formal specification languages, such as LOTOS [10], which also supported by CADP. First, although modeling the DTD in LOTOS is theoretically possible, using LOTOS NT made the development of a formal model practically feasible. Second, because the formal model is easily understandable by the architect, it can serve as a basis for trying alternate designs, i.e. to experiment with complex performance optimizations that would otherwise be discarded as too risky. Last, but not least, the automatic analysis capabilities offered by CADP (e.g. step-by-step simulation, model checking, co-simulation) increased the confidence in the DTD.

The rest of the paper is organized as follows. Section 2 describes the DTD. Section 3 presents the LOTOS NT model of the DTD. Section 4 reports on formal verification of the DTD using CADP. Section 5 reports the co-simulation of the LOTOS NT model and the original C++ model of the DTD. Finally, Section 6 presents our conclusions.

2 Dynamic Task Dispatcher

The joint STMicroelectronics-CEA "platform 2012" project [14] aims at developing a many-core programmable accelerator for ultra-efficient embedded computing. This accelerator includes one or several processor clusters with associated memories and control blocks. We focus on a cluster designed for fine grain parallelism (data and task level).

The underlying programming model is the "ready to run until completion" model, i.e. a task can be divided in several sub-tasks, which, if each sub-task has all the data needed for its completion at the time it is launched, can be executed

[1] `http://vasy.inria.fr/multival`

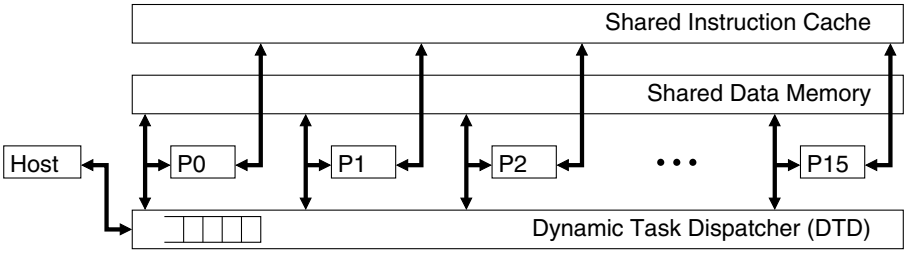


Fig. 1. Global architecture of the cluster

in parallel. As there is no interaction between sub-tasks, the sub-tasks respect the Bernstein conditions [2], and thus can be executed in any order, even in parallel (this might be required to reach the expected performance). One of the routines for sub-task-execution is $dup(void\ *f(int\ i),\ int\ n)$, which replicates n times the execution of the function f (each instance receiving a different index i as argument), and terminates when all the sub-tasks have terminated.

In order for this execution scheme to be efficient, task switching must require only a few cycles and sub-tasks must be allocated at run time to an idle processor. This has several implications on the hardware architecture. First, this cluster is based on a data memory shared by all processors. Thus, even if a sub-task runs on the different processor than its ancestor, it has the same frame pointer and thus an easy access to global variables. Second, all processors share the same instruction cache, lowering the cost of replicating a task on several processors. Lastly, a dedicated hardware block, the Dynamic Task Dispatcher (DTD), is responsible for task selection and launch on the selected processor.

The cluster consists of 16 STxP70 processors, extensible 32-bit microcontrollers with an Harvard architecture (separated data and instruction busses). Communication with the DTD is performed through data accesses on dedicated addresses. The DTD is thus connected, in parallel, to the data bus of each processor. A processor will use a *store* operation to ask the DTD to dispatch a task and a *load* operation when willing to execute a new task. Figure 1 shows the overall architecture, designed as globally asynchronous, locally synchronous (GALS) system: Even if all the processors run at the same clock frequency, their clocks may not be synchronized due to physical limitations. Furthermore, due to its complexity, the DTD is not targeted to run at the same clock frequency as the processors.

In order to reduce power consumption, inactive processors are kept in idle mode and are woken up by the DTD using an asynchronous wakeup signal. After wakeup, a processor immediately issues a *load* to a memory mapped address of the DTD. The answer to a *load* is either a task descriptor, containing the address of a function to execute (in this case, the processor jumps to the address and executes the function), or a special descriptor indicating that there is no more work (in this case, the processor switches to the idle mode). To signal the end of task execution, a processor issues a *load* for a new task.

The implementation of *dup*() first issues a *store* to ask for a task to be dispatched, and then enters a loop, which starts by issuing a *load*. The response is a task descriptor (in this case, the processor executes the task — a processor is guaranteed to execute one instance of the function it asked to replicate), a special descriptor indicating that there are no more instances to execute but some instances executing on other processors are not yet terminated (this case is called active polling), or a special descriptor indicating that all the sub-task have been executed (in this case, the processor can leave the loop and go on executing the calling task). The cluster supports three levels of nested tasks per processor, which is enough for the forcasted applications and is not too expensive in term of silicon area.

The DTD also has an interface to handle the main tasks requests issued by the host processor (application deployment on the accelerator). This interface is connected to a queue and as soon as there is a task to execute in this queue and an idle processor, the task is assigned to the processor and removed from the queue.

Figure 2 shows a sub-task execution scenario using three processors. The processor P0 requests the execution of four instances of the sub-task foo(). Processor P0 is assigned the execution of the sub-task with index 3, processors P1 and P2 are awakened and assigned the execution of the sub-tasks with respective indexes 2 and 1. As execution on processor P2 terminates, P2 is assigned the execution of the sub-task with the last index, 0. When the processor P0 finishes its execution, it is first informed that it has to wait for the completion of sub-tasks instances (LD_RSP (WAIT_SLAVE)). When asking once more after all sub-tasks have been executed, P0 is informed about the completion (LD_RSP (DONE)).

3 Formal Model of the DTD

We formally modeled the DTD using LOTOS NT [3,7], a variant of the E-LOTOS [11] standard implemented within CADP. LOTOS NT combines the best of process-algebraic languages and imperative programming languages: a user-friendly syntax common to data types and processes, constructed type definitions and pattern-matching, and imperative statements (assignments, conditionals, loops, etc.).[2] LOTOS NT is supported by the lnt.open tool, which translates LOTOS NT specifications into labeled transition systems (LTSs) suitable for on-the-fly verification using CADP.

3.1 Design Choices

From the DTD point of view, all the interfaces (with the host, the memory, and the processors) evolve in parallel: hence, an unconstrained state space exploration would lead to a state space explosion. Furthermore, the applications running on processors must respect some rules that are embedded in the

[2] We use the notation "**when** C_1 **then** B_1 ... **else when** C_n **then** B_n **end when**" as syntactic sugar for "**if** C_1 **then** B_1 ... **elsif** C_n **then** B_n **else stop end if**".

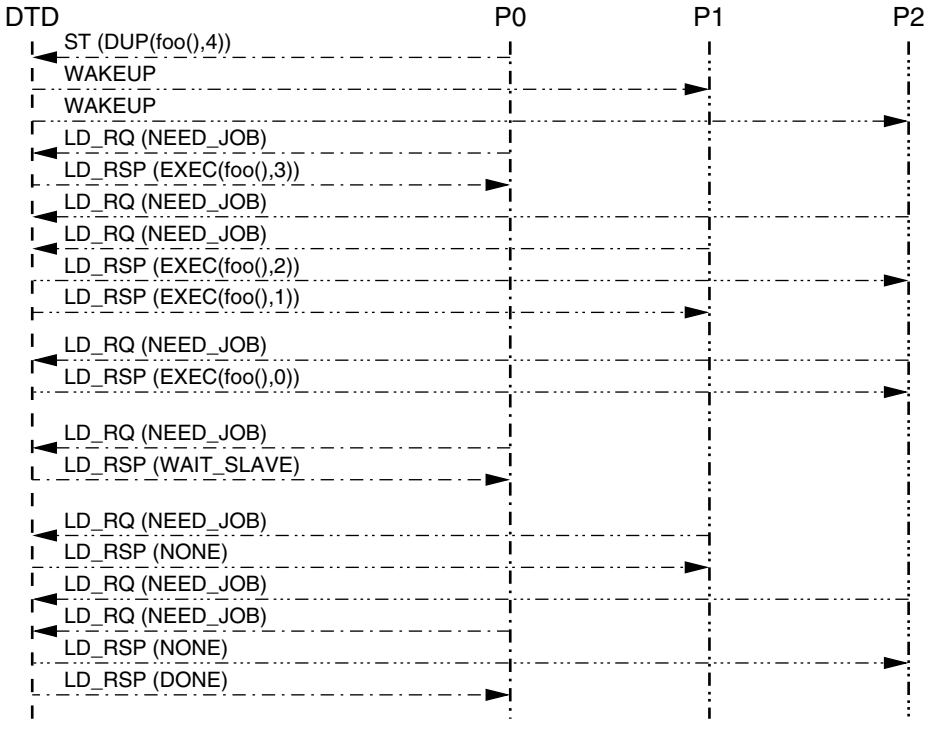


Fig. 2. Sub-task execution scenario

programming model such as the number of nested tasks and order of transactions on the interface. Modeling these rules in the DTD model would be artificial. For all these reasons, we have chosen to abstract the application to typical scenarios, running on abstracted processors, and to perform our verifications on each identified scenario.

The classical way of verifying a hardware block is to run massive simulations. For a block like the DTD, these simulations mean executing several scenarios. These simulations rely on the event scheduler of the simulator. Precise hardware simulations of the whole system are expensive in time and some abstractions are used, which imply that the resulting scheduling may not be the same as the real one. Even if we restrict the verification of our formal model to a set of scenarios, we explore *all* the scheduling possibilities for each scenario. Furthermore, we are able to use model checking, which is impossible for standard simulations.

We decided to model everything, hardware (both the DTD and the processors), applications, and software routines (namely *dup*()) using LOTOS NT processes, because only the code inside a LOTOS NT process has access to the gates and can synchronize with other processes. For example, it is mandatory to define a *dup*() as a sequence of three rendezvous, namely a store, a load request, and a load response.

The representation in an asynchronous language of events taken into account simultaneously was a modeling challenge. Indeed, the DTD is a classical synchronous hardware block, scanning its inputs at each cycle and computing the relevant outputs. Hence, the decisions taken by the DTD are not based on a response to a single input but on the totality of all inputs. We did not want to artificially synchronize on a global clock, so we used a multi-phase approach: an input is, asynchronously, taken into account by modifying the internal vector state S_i, and outputs are issued according to S_o. The outputs are computed, asynchronously, by scanning the vector state S_i, updating the vector state S_o by a decision clause. This clause may include a rendezvous on a gate, which can be seen as clock for this decision function in a synchronous design. This rendezvous also prevents there being non-determinism in the generated LTS. This approach enables interleaving of synchronisation in the independent interfaces of the model because the model is never blocked waiting for a synchronization and parallel parts of the model evolve atomically.

The main difference between this approach and that proposed for integrating a synchronous automaton in an asynchronous environment [8] is that we need to aggregate several asynchronous events into a single synchronous event, whereas in [8] each asynchronous message is decomposed into a set of synchronous signal changes.

Figure 3 presents the code of a simple arbiter respecting the rules presented and the associated LTS. This arbiter has 2 interfaces A and B, the states of which are recorded in the variables state_A and state_B. Each interface evolves by the rendezvous on gate I followed by the rendezvous on gate O. The first two **when**-clauses deal with the first rendezvous and modification of the state, while the last two clauses deal with the second rendezvous, according to the computed state. The middle clause is the decision function which updates the state. This clause issues a rendezvous on gate D. The priority given to the A interface can be seen on state 4 of the LTS.

3.2 Modeling the Dynamic Task Dispatcher Hardware[3]

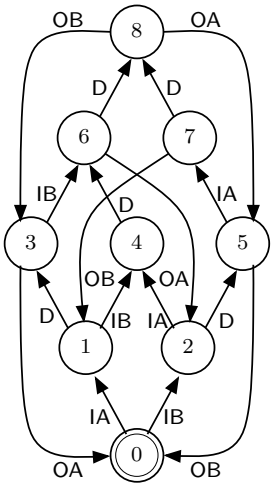
From the DTD point of view, the state of a processor can be unknown (before the processor signals it has booted), idle (in the idle mode), neutral (executing a top-level task), master (having caused a dispatch of sub-tasks by calling *dup*()), or slave (executing a sub-task dispatched by another processor). In the last case, the DTD has to keep a reference to the corresponding processor having called *dup*(). Due to the nested task mechanism, the processor state has to be kept in a stack-like structure of fixed depth.

Additionally, we have to record the state of the interface of each processor. The state of the interface of a processor is used to propose the relevant rendezvous. For example, the running state of the interface is used when the processor executes a task or a sub-task, so that the interface can accept a load signaling the

[3] The DTD model is considered confidential by STMicroelectronics and cannot be presented in more detail.

```
process Arbiter [IA, OA, IB, OB, D: none] is
  var state_A, state_B: Nat in
    state_A := 0; state_B := 0;
    loop select
      (* handling first rendezvous ("input") *)
        when state_A == 0 then IA; state_A := 1 end when
      [] when state_B == 0 then IB; state_B := 1 end when
      (* decision function *)
      [] when state_A == 1 then state_A := 2 else
        when state_B == 1 then state_B := 2 end when;
        D (* marking the decision *)
      (* handling second rendezvous ("output") *)
      [] when state_A == 2 then OA; state_A := 0 end when
      [] when state_B == 2 then OB; state_B := 0 end when
    end select end loop
end var end process
```

(a) LOTOS NT specification



(b) LTS

Fig. 3. Example of an arbiter

end of task execution. Each rendezvous affects only the state of the corresponding interface: thus, all interfaces can change independently of the others. DTD decisions are based on (and modify) all the interface states and processor states.

The model of the DTD is thus described by a LOTOS NT process Dtd executing an infinite loop containing:

- For each processor, several guarded clauses dealing with its interface. Each of these clauses handles a rendezvous with the processor and updates the variables representing the state of the processor interface. A clause also deals with the communication with the host processor, filling a queue with task requests. These blocks of code implement the connection between the asynchronous communication scheme and the synchronous decision function.
- Several clauses to achieve the dispatches requested by the tasks executing on the processor and to launch tasks requested by the host. This corresponds to the function executed by the DTD on each cycle.

The communication between the DTD and processor n is modeled using four gates: WAKEUPn, LD_RQn (load request), LD_RSPn (load response), and STn (store, considered to be atomic).

The number of processors impacts the internal structure of the DTD, mainly because the arbitration is based on the global state vector, and not on local properties of some part of it. Hence, a generic model of the DTD parameterized by the number of processors would be complex. Instead, we choose to develop a model generator that takes a number of processors and generates the corresponding LOTOS NT process Dtd. This development was facilitated by the structure of the model. The part dealing with a processor interface is just replicated using

```
type PC_T is pc_1, pc_2, pc_3 with "==", "!=" end type

process Execute [ST, LD_RQ, LD_RSP, MSG: any]
                (j: Job_Desc_T, inout S:Job_Desc_Stack_T) is
  var pc: PC_T, index: Nat in
    pc := get_PC (j);
    case pc in
      pc_1 -> MSG ("pc_1");
              Dup [ST, LD_RQ, LD_RSP, MSG] (pc_3, 4, EXEC(pc_2, -1), !?S)
    | pc_2 -> MSG ("pc_2: Master after Dup")
    | pc_3 -> MSG ("pc_3: slave with index ", get_Index (j))
    end case
end var end process
```

Fig. 4. Scenario 2 for four processes: creation of four sub-tasks

some naming conventions for the variables and gates used in this blocks. The decision part of the model requires changes to some loop bounds and extension of the number of case clauses of some **select** statements. The resulting code size ranges from 1020 lines (170 lines per processor and 230 lines for the decision part) for four processors to 8530 lines (376 lines per processor and 2328 lines for the decision part) for 16 processors.

3.3 Modeling Applications and Processors

First, we define an enumerated type, called PC_T representing the addresses of the task functions. To circumvent a limitation of the LOTOS NT compiler, which rejects some non-tail recursive calls, we include a call-stack in the processor model; this call-stack is passed by reference (mode **inout**) to the processes Execute and Dup implementing the execution of the tasks.

The execution of a task function is modeled by a simple process, called Execute that is mainly a switch between the various values of PC_T, as shown in Figure 4.

Dup adds the continuation cont (the task function to be executed at the end of the sub-task) to the stack, performs the store operation, and exits. When Dup exits, so does the calling Execute process. Then Processor requests a new sub-task. After termination of all sub-tasks, Processor calls Execute to execute the continuation, which is removed from the stack. The corresponding LOTOS NT code is shown in Figure 5.

4 Formal Analysis of the DTD

We used the CADP toolbox [4] to generate the LTSs corresponding to twelve scenarios each for four and six processors. Let N be the number of available processors. Scenario 1 defines a set of more than N tasks, which can be executed in parallel. The other scenarios all contain calls to $dup()$, the simplest one being scenario 2 (see also Figure 4). Scenario 2 defines one main task that forks N sub-tasks; scenario 2_1 adds to scenario 2 more sub-tasks and scenario 2_2 adds to

```
process Dup [ST, LD_RQ, LD_RSP: any]
            (pc: PC_T, count: Int, cont: Job_Desc_T, inout stack: Job_Desc_Stack_T) is
  stack := push_job_desc (cont, stack);
  ST (DUP(pc, count))
end process

process Processor [ST, LD_RQ, LD_RSP, WAKEUP: any] is
  var stack: Job_Desc_Stack_T := nil in
    ST(BOOT);
    loop
      WAKEUP;
      loop main_loop in var j: Job_Desc_T in
        LD_RQ (NEED_JOB);
        LD_RSP (?j);
        case j in
          var npc: PC_T, index: Int in
            EXEC(npc, index) -> Execute [ST, LD_RQ, LD_RSP, MSG] (j, !?stack)
          | WAIT_SLAVE -> null
          | DONE ->(* all slaves terminated, pop the continuation *)
              if (is_stack_empty(stack)) then
                break main_loop
              else
                j := head_stack(stack); stack := POP_JOB_DESC(stack);
                Execute [ST, LD_RQ, LD_RSP, MSG] (j, !?stack)
              end if
          | NONE -> break main_loop
        end case
      end var end loop
  end loop end var end process
```

Fig. 5. Stack-based implementation of Dup and Processor

scenario 2 two other main tasks that do not fork sub-tasks. Scenario 3 uses nested calls of *dup*(): a main task forks sub-tasks that also fork, the total number of tasks and sub-tasks being greater than N. Scenarios 3_1 and 3_2 change the number of sub-tasks for each level of invocation, and scenario 3_3 adds to scenario 3 two other main tasks that do not fork sub-tasks. The main task of Scenario 4 invokes *dup*() twice consecutively, each time forking more than N sub-tasks; scenario 4_1 just forks more sub-tasks at each invocation of *dup*() than scenario 4. Lastly, scenario 5 consists of two main tasks, each invoking *dup*().

Table 1 summarizes the state space sizes and generation times using a computer with a 2.8 GHz processor and 120 GB of RAM. For six processors, LTS generation was possible for only five scenarios, and for more processors, even the generation of the smallest scenario ran out of memory. For even smaller scenarios (only two tasks in scenario 1, or a duplication to only two processors in application 2), the LTS can be visualized step-by-step and checked manually.

Table 1. LTS sizes, as well as generation and verification times (in seconds)

N (# proc)	scenario	size		generation time	verification time			
		states	transitions		prop. 1	prop. 2	prop. 3	prop. 4
4	1	664,555	2,527,653	30.62	2917.46	2766.66	1.21	3379.43
	2	28,032	91,623	2.46	.19	.55	.38	.48
	2_1	73,984	255,391	3.75	.26	1.32	.76	.94
	2_2	920,649	3,537,763	39.44	.79	421.02	6.29	429.11
	3	168,466	557,363	8.13	.28	1.43	1.05	1.39
	3_1	1,445,922	5,204,671	69.07	.94	12.52	8.01	13.19
	3_2	665,546	2,387,195	27.87	.59	6.21	3.42	5.17
	3_3	4,435,309	17,328,979	229.02	2.63	482.89	32.32	476.42
	4	63,760	211,579	3.90	.22	.99	.55	.72
	4_1	168,288	586,539	7.31	.33	2.49	1.32	1.69
	5	181,170	596,022	8.82	.29	1.81	1.18	2.43
	5_1	1,626,933	5,989,205	63.52	1.27	20.07	10.21	37.59
6	2	4,998,344	24,324,439	312.85	4.83	339.92	108.24	168.97
	2_1	14,778,488	74,826,343	970.13	16.73	1551.16	752.54	545.56
	4	12,696,086	62,482,651	1048.09	9.97	843.62	404.25	374.80
	4_1	37,090,190	189,595,795	3049.07	33.85	3479.69	1430.14	1605.42
	5	97,297,953	489,846,494	9022.89	62.70	6405.57	2170.91	5344.10

To gain confidence in our model, we included assertions that, if violated, would yield an ERROR transition. We checked for all scenarios that the generated LTS did not contain such an ERROR transition.

To formally verify the correct execution of the different scenarios, we expressed some properties using the MCL language [13]. The ability to capture the number of a processor in one transition label proved to be crucial to expressing a property in a concise and generic way.[4]

A first formula expresses that each scenario is acyclic, i.e. from each state, a terminal state without outgoing transitions is eventually reached:

$$\mu\ X\ .\ [\ \mathbf{true}\ ]\ X$$

The set of states satisfying this fix-point formula is computed iteratively, starting with $X = \emptyset$: Initially, "[**true**] $\emptyset$" is satisfied by states without outgoing transitions, and iteration k adds to X those states from which a deadlock can be reached in k steps.

Unfortunately, this property does not hold for all scenarios with a *dup*() operation, because the master processor stays in its state after receiving a WAIT_SLAVE. Indeed, the third block of messages in Figure 2 (i.e. "LD_RQ (NEED_JOB)" followed by "LD_RSP (WAIT_SLAVE)") might be repeated an arbitrary number of times. However, under the hypothesis that each slave always terminates, such a cycle is executed a finite number of times. Thus, cycles of this form should not be considered a problem, and the property must be refined,

[4] This required renaming (on the fly) all gates to extract the number of the corresponding processor, e.g., STn has to be renamed into the pair "ST !n".

for instance by requiring that only cycles of this form are permitted, i.e. that the system inevitably reaches either a deadlock or gets stuck in a cycle of the permitted form (the formula "< **true*** . φ >@" is satisfied by all states of a cycle containing a transition with a label of the form φ):

$\mu\ X\ .\ (\ [\ \textbf{true}\ ]\ X\ \textbf{or}\ (\textbf{exists}\ y{:}\mathsf{Nat}\ .\ \langle\textbf{true}^*\rangle\{\mathsf{LD\_RSP}\ !y\ !"\mathsf{WAIT\_SLAVE}"\}\rangle@))$

A second formula expresses that, after waking up a processor, the DTD eventually tells the processor that there is no more work left, i.e. each WAKEUPx is eventually followed by "LD_RSPx !NONE" (where x is a processor number):

$[\ \textbf{true}^*\ .\ \{\mathsf{WAKEUP}\ ?x{:}\mathsf{Nat}\}\ ]\ \mathsf{inevitable}(\{\mathsf{LD\_RSP}\ !x\ !"\mathsf{NONE}"\})$

Note how the number x of the processor woken up is extracted from a transition label by the first action predicate "{WAKEUP ?x:Nat}" and is used subsequently in the property. The predicate "inevitable(B)" expresses that a transition labeled with B is eventually reached from the current state. It can be defined in MCL by the following macro definition:

macro inevitable(B) =
$\mu\ X\ .$
$\Big($< **true** > **true** **and**
$([\textbf{not}(B)]\ X\ \textbf{or}\ (\textbf{exists}\ y{:}\mathsf{Nat}\ .\ \langle\textbf{true}^*\rangle\{\mathsf{LD\_RSP}\ !y\ !"\mathsf{WAIT\_SLAVE}"\}\rangle@))\Big)$
end_macro

As for the first property, the definition of inevitable ignores any (spurious) cycles corresponding to a master processor waiting indefinitely for the slave processes to terminate.

A third formula expresses that each call to *dup*() executes to completion, i.e. each "STx !DUP" is eventually followed by "LD_RSPx !DONE":[5]

$[\ \textbf{true}^*\ .\ \{\mathsf{ST}\ ?x{:}\mathsf{Nat}\ !"\mathsf{DUP}"\}\ ]\ \mathsf{inevitable}(\{\mathsf{LD\_RSP}\ !x\ !"\mathsf{DONE}"\})$

A final formula expresses that each task sent by the host application is executed exactly once, i.e. each "HOST !c" (c being the task to be executed) is eventually followed by a transition of the form "LD_RSP !x !c", but cannot be followed by a sequence containing two transitions of the form "LD_RSP !y !c" (x and y being processor numbers, and c being the task received previously from the host processor):[6]

$[\ \textbf{true}^*\ .\ \{\mathsf{HOST}\ ?c{:}\mathsf{String}\}\ ]$
$\Big(\mathsf{inevitable}(\{\mathsf{LD\_RSP}\ ?x{:}\mathsf{Nat}\ !c\})\ \textbf{and}\ \big[(\textbf{true}^*\ .\ \{\mathsf{LD\_RSP}\ ?y{:}\mathsf{Nat}\ !c\})\{2\}\big]\ \textbf{false}\Big)$

In this formula, the expression "$(\textbf{true}^*\ .\ \{\mathsf{LD\_RSP}\ ?y{:}\mathsf{Nat}\ !c\})\{2\}$" characterizing transition sequences that contain exactly two repetitions of a sequence of the form "**true*** . {LD_RSP ?y:Nat !c}".

[5] This property requires an additional renaming operation to suppress the parameters of the DUP operation, i.e. to rename "STn !DUP (...)" to "ST !n !DUP".

[6] This property requires an additional renaming operation, namely to rename "LD_RSPn !EXEC(c, -1)" to "LD_RSP !n !c".

Using the EVALUATOR 4 model checker [13], we verified these properties in about ten hours for all 17 scenarios for which we had generated the LTS (see Table 1 for details).

Because our formal LOTOS NT model is simpler to modify than the one used by the architect, we also explored different architectural choices and optimizations. In order to get better performance, we wanted to avoid a processor from going into idle mode when a task needed to be executed. Due to timing constraints in the decision process of the real hardware, a slave processor that terminates a sub-task can only be assigned immediately to another sub-task from the same master processor. When no more sub-tasks are available, the slave processor goes in the idle state even if there are pending tasks to execute (main tasks or sub-tasks from another master processor). We proposed that, when terminating a sub-task, a processor asks the DTD a second time for a task to execute. This answer to this second request would be treated, in the real hardware, by a decision process different from the one involved in the first request and should meet the timing constraints. We checked on our model that this behavior would lead to a correct execution scheme, before the architect made the modification.

5 Co-simulation of the C++ and LOTOS NT Models

The DTD has been designed by the architect directly as a C++ model suitable for high level synthesis tools such as CatapultC[7] or the Symphony C compiler[8]. Therefore, this model follows the synchronous approach commonly applied in the hardware design community. In this approach, a hardware block is represented as a function $f : \mathit{inputs} \times \mathit{state} \rightarrow \mathit{outputs} \times \mathit{state}$ that is called on each clock cycle to evaluate its inputs and to compute the outputs and the new internal state to be used in the next clock cycle.

The C++ model of the DTD comes with a clock-based simulation environment providing abstractions of the host processor, the cluster processors, and the software executing on them. In order to assess the correctness of the C++ model (and thus the generated hardware circuit), we experimented with the co-simulation of the C++ and LOTOS NT models, using the EXEC/CÆSAR framework [9]. Practically, we added the LOTOS NT process Dtd (i.e. the model of the DTD without its environment) to the simulation environment coming with the C++ model. Keeping also the C++ model of the DTD ensured that both models were exposed to the same stimuli, enabling us to crosscheck both models, in particular that both models behave similarly. This differs from classical co-simulation environments where a part of a design is replaced by a model not depicted at the same level of abstraction, such as an *Instruction Set Simulator* of a processor inserted in the simulation of a full *System On Chip* with peripherals depicted in a hardware description language.

In some sense, this co-simulation is similar to model-based conformance testing, as for instance with TGV [12] or JTorX [1]. Taking as input a model and a

[7] http://www.mentor.com/esl/catapult/overview

[8] http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx

test purpose, TGV computes a test case that, when used to test an implementation, enables conformance of the implementation to the model to be checked: without a test purpose, our co-simulation simply checks the conformance of each step in an execution. Contrary to JTorX, our approach does not require an explicit representation of the model, which avoids the state explosion problem and enables the co-simulation of the DTD for 16 processors (for which we could not generate the LTS).

The main challenge was the combination of asynchronous event-based LOTOS NT model with a synchronous clock-based C++ model and simulation environment. Indeed, in one single clock cycle, several inputs to the DTD might change, and it might also be necessary to change more than one output: thus, a single simulation step of the C++ model might require several events (i.e. rendezvous synchronizations) in the LOTOS NT model. To further complicate matters, the number of events corresponding to a single clock cycle is not known in advance, because it depends on the current state and inputs.

Before presenting our approach to driving an asynchronous model within a synchronous simulation environment and the results of our experiments, we briefly recall the principles of the EXEC/CÆSAR framework.

5.1 Principles of the EXEC/CÆSAR Framework

In the EXEC/CÆSAR framework, a LOTOS NT model interacts with its simulation environment only by rendezvous on the visible gates. Practically, for each visible gate, the simulation environment has to provide a C function, called a *gate function*; offers of the rendezvous are passed as arguments to the gate function (in a nutshell, offers sent from the LOTOS NT model to the environment are passed by value, and offers received from the environment are passed by reference). Each gate function returns a boolean value, indicating whether or not the simulation environment accepts the rendezvous.

Using the CÆSAR compiler [6,5], a LOTOS NT model is automatically translated into a C function f, which tries to advance the simulation by one step. In each state, f first determines the set of rendezvous permitted by the LOTOS NT model; if this set is empty, f signals a deadlock, otherwise it iterates on the elements of the set, calling the corresponding gate functions with appropriate parameters. As soon as one rendezvous is accepted by the environment, the model performs the corresponding transition and moves to the next state. If none of the rendezvous is accepted, f returns with an indication that the state has not changed; this feature enables the simulation environment to compute the set of all rendezvous possible in the current state of the LOTOS NT model; calling f once more then enables one of these rendezvous to be accepted.

5.2 Approach

To integrate the asynchronous LOTOS NT model into the synchronous C++ simulation environment, we took advantage of the feature of EXEC/CÆSAR mentioned above to compute the set of all enabled rendezvous. We also

exploited the fact that, as usual for hardware circuits, input and outputs can be distinguished by the gate of the rendezvous: the gates ST, LD_RQn, and HOST represent inputs of (i.e. signals received by) the DTD, whereas the gates LD_RSPn and WAKEUPn represent outputs of (i.e. signals sent by) the DTD. Furthermore, we used the fact that any output of the DTD is always the reaction to (a set of) inputs. Last but not least, we relied on the modeling style, in particular the independence of the different interfaces of LOTOS NT model of the DTD. Indeed, for a set of actions (only inputs or only outputs) that may occur in the same clock cycle, the modeling style ensures the confluence of the execution of the actions in the set, i.e. when the LOTOS NT model of the DTD executes such a set of actions, all orderings lead to the same state. Thus, one can arbitrarily choose one ordering.

Concretely, to simulate the equivalent of one clock cycle of the synchronous C++ model, we execute the following steps.

- Iterate over all proposed rendezvous to compute the set of all enabled outputs of the LOTOS NT model. If this set is different from the set of outputs produced by the C++ model (since the last clock), signal an error.
- Accept all outputs in the set once. If an output is enabled more than once, signal an error.
- Iterate over all proposed rendezvous to compute the set of all enabled inputs of the LOTOS NT model. If this set does not include all inputs to be given to the C++ model, signal an error.
- For all inputs given to the C++ model, provide them once to the LOTOS NT model.
- Accept the rendezvous marking the execution of the decision function.

If we apply this approach to the arbiter example presented in Figure 3, the output signals are OA and OB, input signals are IA and IB, and the decision making signal is D. In a co-simulation, the behavior of the model will not cover the full LTS as an output is always accepted before the next input. For example, in state 3, the input transition $3 \rightarrow 6$ cannot be taken, due to the output transition $3 \rightarrow 0$; this implies that transition $6 \rightarrow 8$ is never taken. Because also transition $5 \rightarrow 7$ cannot be taken, states 7 and 8 are unreachable. Thus, co-simulation obviously explores only a sub-set of the LTS.

5.3 Results

Using the EXEC/CÆSAR framework, we co-simulated the LOTOS NT model of the DTD for 16 processors with the architect's C++ model, using the architect's simulation environment for stimuli generation. After a ramp-up phase mainly devoted to fine-tuning which signal should be considered in which clock phase and dealing with C/C++ mangling, we were able to run the first scenarios. Being clock-based, the simulation environment imposes the scheduling of the signals; this corresponds to selecting a path in the LOTOS NT model.

For some applications, we found a difference in the choices made by the LOTOS NT and the C++ models. This revealed that the decision part of the

two models was not written in the same way. For implementation reasons, the architect's C++ model uses a decision tree, while the LOTOS NT model uses an iterative approach. This highlighted, once again, that a natural-language specification is subject to different interpretations. We modified and re-validated the LOTOS NT model to fit the decisions made by the C++ model.

Although clock-based, the simulation environment should be considered only as cycle-approximate, i.e. only the interaction between the DTD and the processors are precisely modeled, whereas execution time of both memory latency and instruction execution in processors is not modeled precisely. The LOTOS NT model is insensitive to the latter execution times, as it proposes *all* interleavings. Because important properties have been formally verified on LOTOS NT model, and the C++ model behaves as the LOTOS NT model on the execution scheme proposed by the simulation environment, we gained confidence in the fact that the C++ model should have correct behavior in cases not proposed by the simulation environment, which is clearly an added value compared to solely simulation-based validation.

6 Conclusion

We illustrated that LOTOS NT, a formal modeling language based on process algebra, is well-suited for modeling, design-space exploration, analysis, and co-simulation of a complex industrial hardware circuit in an asynchronous multiprocessor environment. This increased the confidence in the design and enabled the integration of an optimization that might otherwise have been judged too risky. Although all this would certainly have been possible using a classical formal specification language or other formal methods, we found that using LOTOS NT helped in obtaining the model and communicating with the architect, and might be an interesting addition to the design flow.

This work points to several research directions. First, the case study poses a challenge of using more elaborate and/or prototype state space exploration techniques (e.g. distributed, compositional, and on-the-fly verification, or static analysis for state space reduction) to handle larger scenarios. Second, it would be interesting to consider a more general version of the DTD where each processor would, after boot, declare its instruction-set extensions, and the *dup*() operation would also specify the required instructions.

Acknowledgments. We are grateful to Michel Favre (STMicroelectronics) for discussions about the architecture of the DTD and to Radu Mateescu (INRIA) for help with the expression of correctness properties in MCL.

References

1. Belinfante, A.F.E.: JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
2. Bernstein, A.J.: Analysis of Programs for Parallel Processing. IEEE Transactions on Electronic Computers EC-15(5), 757–763 (1966)

3. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.1). INRIA/VASY, pages 117 (December 2010)
4. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
5. Garavel, H., Serwe, W.: State Space Reduction for Process Algebra Specifications. Theoretical Comput. Sci. 351(2), 131–145 (2006)
6. Garavel, H., Sifakis, J.: Compilation and Verification of LOTOS Specifications. In: Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification. IFIP, pp. 379–394. North-Holland, Amsterdam (1990)
7. Garavel, H., Sighireanu, M.: Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In: Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS 1998, pp. 187–230 (May 1998) CWI. Invited lecture
8. Garavel, H., Thivolle, D.: Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In: Păsăreanu, C.S. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 241–260. Springer, Heidelberg (2009)
9. Garavel, H., Viho, C., Zendri, M.: System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. Springer International Journal on Software Tools for Technology Transfer 3(3), 314–331 (2001)
10. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization (September 1989)
11. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization (September 2001)
12. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms — A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. Springer International Journal on Software Tools for Technology Transfer 7(4), 297–315 (2005)
13. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
14. STMicroelectronics/CEA. Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology (November 2010), `http://www.2parma.eu/images/stories/p2012_whitepaper.pdf`

Transforming SOS Specifications to Linear Processes

Frank P.M. Stappers[1], Michel A. Reniers[2], and Sven Weber[3]

[1] Department of Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
[2] Department of Mechanical Engineering, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
[3] Department for Architecture and Platform, ASML,
P.O. Box 324, NL-5500 AH Veldhoven, The Netherlands

Abstract. This paper describes an approach to transform Structural Operational Semantics, given as a set of deduction rules, to a Linear Process Specification. The transformation is provided for deduction rules in De Simone format, including predicates. The Linear Process Specifications are specified in the syntax of the mCRL2 language, that, with help of the underlying (higher-order) re-writer/tool-set, can be used for simulation, labeled transition system generation and verification of behavioral properties. We illustrate the technique by showing the effect of the transformation from the Structural Operational Semantics specification of a simple process algebra to a Linear Process Specification.

1 Introduction

The behavior of a *system* can be analyzed in various ways. It can be achieved by observing output from simulations, or by examining the behavioral descriptions (*e.g.*, code of a controller). To perform such an analysis, one always requires syntax (the way to denote behavior), semantics (the way in which grammatically correct behavior is executed) and a relationship between the two.

One way to describe the formal execution behavior of a system, is to use *Structural Operational Semantics* (SOS) [33]. Here, semantics is assigned to syntax, by means of deduction rules that describe the allowed set of actions of a piece of syntax. Unfortunately, there are hardly any suitable automated transformations from SOS specifications, along with a syntactical instance, to languages that can be subjected to formal analysis.

In this paper, we address this gap by formulating a systematic approach by which the deduction rules specified in SOS, along with the signature of the syntax, are transformed into a symbolic representation of a labeled transition system, called a Linear Process Specification (LPS) [6,20]. The LPS can later be subjected to formal analysis (*e.g.*, simulation, explicit labeled transition system generation, and verification). We restrict the deduction rules to the De Simone format [18].

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 196–211, 2011.
Springer-Verlag Berlin Heidelberg 2011

We have chosen LPS as a target formalism, because it (i) has a mathematical representation that strongly relates to deduction rules in SOS and (ii) can be directly implemented in the mCRL2 language [22,27]. In fact, LPS serves as a backbone for the representation and manipulation of behavioral models in the mCRL2 tool-set. Since this tool-set facilitates a higher-order term rewrite system, a transition generator and other transformation tools, we are able to exhaustively explore the state space and conduct formal behavioral analysis.

The framework aims at the transformation of formal behavioral specifications to specifications that are suitable for analysis, *e.g.*, simulation and model-checking. The technique can be, and is, used [35], when prototyping formal (domain specific) languages, to investigate behavior dictated by the underlying operational semantics or to automate translations of formal languages towards the mCRL2 tool-set.

Outline. Section 2 describes the preliminaries on SOS and LPS. Section 3 describes the transformation of the signature and SOS of a language to an LPS. Section 4 provides a small but nevertheless illustrative example. Section 5 discusses discrepancies between the presentation and implementation. In Section 6, we discuss extensions of the framework such as predicates. In Section 7, we position this work. Section 8 concludes and elaborates on future work.

2 Preliminaries

2.1 Structural Operational Semantics

Structural Operational Semantics (SOS) defines the possible actions that a piece of syntax can perform. SOS is typically represented by a transition system specification (TSS) [8]. The syntax for which the semantics is defined, is represented by a signature. A signature fixes the composition operators and their corresponding arities, where a function with arity zero represents a constant. We assume a set of variables $\mathcal{V}$ and a set of action labels $\mathcal{A}$. Note, that the definitions of signature and transition system specification as used in this paper are restricted to signatures with a single sort and transition system specifications with a single transition relation symbol.

A *signature* Σ is a collection of function symbols together with their arities. The arity of a function symbol $f \in \Sigma$ is denoted $ar(f)$. The collection of *terms* over signature Σ, denoted $\mathcal{T}(\Sigma)$, is the smallest set such that (i) a variable $x \in \mathcal{V}$ is a term, and (ii) if $t_1, \ldots, t_n$ are terms and $f \in \Sigma$ is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term. The set of *closed terms* over signature Σ, denoted $\mathcal{C}(\Sigma)$, is the set of all terms over Σ in which no variables occur. The *variables* that occur in a term p are denoted by $vars(p)$. A *transition formula* is of the form $p \xrightarrow{l} p'$ for $p, p' \in \mathcal{T}(\Sigma)$ and $l \in \mathcal{A}$.

A *transition system specification* (TSS) is a tuple $(\Sigma, \mathcal{D})$ where Σ is a signature and $\mathcal{D}$ is a set of deduction rules. A deduction rule is of the form $\dfrac{H}{C}$ where

H is a set of transition formulas, called the set of *premises* and C is a transition formula, called the *conclusion*.

To illustrate our technique, we only consider TSSs that consist of deduction rules of a specific form; we restrict to TSSs in the De Simone format [18]. A TSS $(\Sigma, \mathcal{D})$ is in De Simone format, if every deduction rule $d \in \mathcal{D}$ complies to the following form:

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{l} t}[Cond_d]$$

where all of $x_1, \ldots, x_{ar(f)}$ and y_i, for $i \in I$ are distinct variables, $f \in \Sigma$, $I \subseteq \{1, \ldots, ar(f)\}$, and t is a process term that only contains variables from $\{x_j \mid j \notin I\} \cup \{y_i \mid i \in I\}$ and does not have repeated occurrences of variables, l_i's and l are labels and $Cond_d$ is a condition on the labels of the premises and the label of the conclusion. A TSS defines a set of transitions, a so-called transition relation; see, *e.g.*, [2,32] for formal definitions thereof.

2.2 (Simplified) Linear Process Specifications

In this paper we transform a TSS to an LPS. Informally, an LPS consists of a signature, variable declarations, a collection of data equations, action declarations, a linear process equation, and an initialization. An LPS can be viewed as a symbolic representation for (possible infinite) labeled transition systems. A formal definition of a Linear Process Specification and its components can be found in [22].

A *signature* is a triple $(\mathcal{S}, \mathcal{C}, \mathcal{M})$ where

1. $\mathcal{S}$ is a set of *sort names*, a non-empty (possiblly infinite) set of data elements.
2. $\mathcal{C}$ is a set of constructor function declarations of the form $f\colon S_1 \times \cdots \times S_n \to S$ with $S_1, \ldots, S_n, S \in \mathcal{S}$. Constructor functions are functions by which exactly all elements in the sort can be denoted.
3. $\mathcal{M}$ is a set of mapping declarations of the form $f\colon S_1 \times \cdots \times S_n \to S$ with $S_1, \ldots, S_n, S \in \mathcal{S}$. Mapping functions define auxiliary functions to rewrite terms of a sort.

The sets $\mathcal{C}$ and $\mathcal{M}$ are disjoint.

A variable declaration is of the form $x_1, \ldots, x_n\colon S$ where the x_i are variable names and S is a sort name. From the signature and the variable declarations, terms (of a certain sort) can be constructed. A data equation is of the form $p = p'$ where p and p' are terms of the same sort.

A linear process equation (LPE) is an equation of the form:

$$X(d{:}D) = \sum_{i \in I} \sum_{e_i : E_i} c_i(d, e_i) \rightarrow a_i(d, e_i) \cdot X(g_i(d, e_i))$$

where I is a finite index set of summand variables, where for $i \in I$ holds:

- $c_i(d, e_i)$ is a term of sort $\mathbb{B}$ (denoting the set of Booleans) that serves as a Boolean guard to allow actions,

- $a_i(d, e_i) \in \mathcal{A}$,
- $g_i(d, e_i)$ is a term of sort D that denotes the next state,
- e_i and E_i denote a variable name and a sort expression, respectively.

The original definition of an LPE allows more features such as actions with data parameters, time annotations, termination, *etc.*, which are not needed in this paper and are therefore omitted. The initialization is a statement of the form $X(p)$, where p is an (open) term of sort D.

3 Method

We provide a template that transforms a TSS (in the De Simone format) to an LPS. This LPS is described in mCRL2 notation, which is a symbolic description of the transition relation (transition system) described by the TSS. In order to directly implement it as an mCRL2 specification, we sometimes slightly deviate from notations that are common in mathematics, (*e.g.*, when denoting a set comprehension). The framework that we present is restricted to the use of *mCRL2-restrictive* TSSs, as defined below. The method is illustrated by an example in Section 4.

Definition 1 (mCRL2-restrictive TSS). *A TSS is* mCRL2-restrictive *if*

1. *the signature Σ contains finitely many function symbols,*
2. *the set of labels $\mathcal{A}$ is finite,*
3. *the set of deduction rules $\mathcal{D}$ is finite, and*
4. *the conditions of the deduction rules can be represented in mCRL2.*

In Section 6, we discuss possibilities for relaxing some of these restrictions.

3.1 Signature Transformation

For a signature Σ that consists of different function symbols $f_1, \ldots, f_n$, we define a sort $\mathcal{T}$ together with additional constructor, projection and recognizer functions in the mCRL2 language by:

$$\begin{aligned}\textbf{sort}\ \ \mathcal{T}\ =\ \textbf{struct}\quad & f_1(\pi_1 \colon \mathcal{T}, \ldots, \pi_{ar(f_1)} \colon \mathcal{T})?is_{f_1} \\ & \vdots \\ & \mid f_n(\pi_1 \colon \mathcal{T}, \ldots, \pi_{ar(f_n)} \colon \mathcal{T})?is_{f_n};\end{aligned}$$

For terms of this sort, $f_1, \ldots, f_n \in \mathcal{C}$ are the constructor functions. The projection functions $\pi_i \in \mathcal{M}$ are used to retrieve argument i of a function symbol. These functions are defined by the equations $\pi_i(f(x_1, \ldots, x_{ar(f)})) = x_i$ in case $i \leq ar(f)$ and undefined otherwise. The recognizer functions $is_{f_i} \in \mathcal{M}$ facilitate the evaluation whether a term is of a particular form. The equations defining recognizer function is_{f_i} are $is_{f_i}(f_i(x_1, \ldots, x_{ar(f_i)})) = \mathit{true}$ and $is_{f_i}(f_j(x_1, \ldots, x_{ar(f_j)})) = \mathit{false}$ for $i \neq j$. Note, that in mCRL2 the equality for sort $\mathcal{T}$ is denoted by $\approx$. For a detailed description about sorts in the mCRL2 language consider [22].

3.2 Transitions

The structured sort S_{Trans} is introduced to model pairs of a label and a term. We assume that the set of action labels, say $\{a_1, \cdots, a_n\}$, is represented by a sort $\mathcal{A}_{Trans}$.

sort $\mathcal{A}_{Trans} = \mathbf{struct}\ a_1 \mid \cdots \mid a_n;$
sort $S_{Trans} = \mathbf{struct}\ sol(\pi_l \colon \mathcal{A}_{Trans}, \pi_t \colon \mathcal{T});$

The projection functions π_l and π_t are used to retrieve the transition label and process term from a solution, respectively.

We introduce a function R_{Trans} that satisfies the property, for all s, s' and labels l

$$sol(l, s') \in R_{Trans}(s) \qquad \text{iff} \qquad s \xrightarrow{l} s'$$

Since every transition is derivable due to a specific lastly applied deduction rule, this is accomplished by introducing a function $R_d \colon \mathcal{T} \to Set(S_{Trans})$ for each deduction rule d of the TSS. Then, for $\mathcal{D} = \{d_1, \cdots, d_n\}$, the function $R_{Trans} \colon \mathcal{T} \to Set(S_{Trans})$ is defined by means of the single equation

var $p \colon \mathcal{T};$
eqn $R_{Trans}(p) = R_{d_1}(p) \cup \cdots \cup R_{d_n}(p);$

Consider a deduction rule d of the form

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{l} t}[Cond_d(l_{i_1}, \ldots, l_{i_{|I|}}, l)]$$

in the De Simone format, where $I = \{i_1, \ldots, i_{|I|}\}$. The equation that is introduced for R_d is given next, followed by an informal explanation of its structure and the used auxiliary functions. Finally, their formal definitions are provided.

$$\begin{aligned}
\mathbf{eqn}\quad R_d(p) = \{\ & s \colon S_{Trans} \\
\mid\ & is_f(p) \\
\wedge\ & \sigma^t(\pi_t(s)) \\
\wedge\ & \exists_{l_{i_1},\ldots,l_{i_{|I|}} \colon \mathcal{A}_{Trans}}(Cond_d(l_{i_1}, \ldots, l_{i_{|I|}}, \pi_l(s)) \\
& \qquad \wedge\ \textstyle\bigwedge_{i \in I} y_i \in vars(t) \Rightarrow sol(l_i, \mu^t_{y_i}(\pi_t(s))) \in R_{Trans}(\pi_i(p)) \\
& \qquad \wedge\ \textstyle\bigwedge_{i \in I} y_i \notin vars(t) \Rightarrow \exists_{z_i \colon \mathcal{T}} sol(l_i, z_i) \in R_{Trans}(\pi_i(p))) \\
\wedge\ & \textstyle\bigwedge_{j \notin I} x_j \in vars(t) \Rightarrow \mu^t_{x_j}(\pi_t(s)) \approx \pi_j(p) \\
\}; &
\end{aligned}$$

The conjunct $is_f(p)$ states that the rule can only be applied to terms p that are headed by function symbol f. The conjunct $\sigma^t(\pi_t(s))$ states that the target term must have the same structure as the term t from the deduction rule. The third, fourth and fifth conjunct state that labels l_i and terms y_i need to be found such that the condition and premises of the deduction rule are satisfied. Here, the third conjunct states that we require a solution that fulfills the condition.

The fourth and fifth conjunct restrict the possible solutions to those that agree with the substitution for the occurrences of x_i and y_i in t to obtain $\pi_t(s)$. The expression $\mu_x^t(p)$ denotes the term (from p) that is used to instantiate variable x in t. The last condition checks that the substitutions used for the source variables, occurring in the target, are those provided by p.

Check target structure. The resulting target term must be an instance of the term t. We define a function $\sigma^t \colon \mathcal{T} \to \mathbb{B}$ that checks this. If t is of the form x for some variable x then we introduce the following equation:

var $p \colon \mathcal{T}$;
eqn $\sigma^x(p) = \mathit{true}$;

and for t of the form $f(t_1, \ldots, t_{ar(f)})$, for some function symbol f and terms $t_1, \ldots, t_{ar(f)}$, we introduce the equation

var $p \colon \mathcal{T}$;
eqn $\sigma^{f(t_1,\ldots,t_{ar(f)})}(p) = is_f(p) \wedge \sigma^{t_1}(\pi_1(p)) \wedge \cdots \wedge \sigma^{t_{ar(f)}}(\pi_{ar(f)}(p))$;

and auxiliary functions $\sigma^{t_i} \colon \mathcal{T} \to \mathbb{B}$ with their corresponding equations.

Capture conditions. The user of this framework has to introduce functions $Cond_d$ that capture the meaning of the conditions in the deduction rules. This means that applicability is restricted to such conditions that can be captured as Boolean expressions in the mCRL2 syntax.

map $Cond_d \colon \mathcal{A}_{Trans} \times \cdots \times \mathcal{A}_{Trans} \times \mathcal{A}_{Trans} \to \mathbb{B}$;

For practical cases, these functions are easily captured in the mCRL2 data language.

Extract instance of a variable. To retrieve the term that is used to instantiate a variable x in the term t, we introduce a projection function $\mu_x^t \colon \mathcal{T} \to \mathcal{T}$.

In case t is of the form x we introduce the equation

eqn $\mu_x^x(p) = p$;

In case t is of the form $f(t_1, \ldots, t_{ar(f)})$ for some function symbol f and terms $t_1, \ldots, t_{ar(f)}$, we introduce an equation

eqn $\mu_x^{f(t_1,\ldots,t_{ar(f)})}(p) = \mu_x^{t_i}(\pi_i(p))$;

for each term t_i in which x occurs. Additionally we add the auxiliary functions $\mu_x^{t_i} \colon \mathcal{T} \to \mathcal{T}$ and their corresponding equations. Note that we only use μ_x^t in those cases where $x \in vars(t)$. Hence it does not matter that the function μ_x^t is not defined for variables different from x that do not occur in t. Since we only consider t in which every variable occurs at most once, μ_x^t is well-defined.

3.3 Linear Process Transition Generator

Basically, transitions are performed as long as the set of solutions belonging to term p is non-empty. So, we declare process X with the process parameter $p\colon \mathcal{T}$. For each iteration, we select a solution s such that $s \in R_{Trans}(p)$ holds. Then, for s we need to dispatch the transition (*e.g.*, $\pi_l(s)$) and update term p to be $\pi_t(s)$. Putting it all together results in:

$$\textbf{proc}\ X(p\colon \mathcal{T}) = \sum_{s:S_{Trans}} s \in R_{Trans}(p) \rightarrow \pi_l(s) \cdot X(\pi_t(s));$$

To obtain the behavior associated with a particular term p, we consider the process $X(p)$:

$$\textbf{init}\quad X(p);$$

The following theorem expresses the correspondence between the labeled transition systems associated with the closed process term p and the mCRL2 process $X(p)$. A proof of this theorem can be found in [34].

Theorem 1 (Correspondence). *Let $(\Sigma, \mathcal{D})$ be an mCRL2-restrictive TSS in the* De Simone *format. Then for every $p \in \mathcal{C}(\Sigma)$, the labeled transition system associated with p and the labeled transition system associated with $X(p)$ are isomorphic.*

4 Application

To illustrate our approach we consider the process algebra MPT from [4] extended with an interleaving parallel composition operator. Assume a finite set of actions $\mathcal{A} = \{a_1, \ldots, a_n\}$. The signature of this language consists of the nullary function symbol 0, the unary function symbols $\alpha.\_$ (for $\alpha \in \mathcal{A}$, $\_$ denoting the argument), and the binary function symbols $\_ + \_$ and $\_ \parallel \_$. In this section we will use infix notation for the binary function symbols, where *zero*, a_i, *alt*, and *par* represent 0, $a_i.$, $+$, and $\parallel$ respectively.

When applying the signature transformation we get:

$$\begin{aligned}
&\textbf{sort}\ \ \mathcal{T} = \textbf{struct}\ \ zero?is_{zero} \mid a_1(\pi_1\colon \mathcal{T})?is_{a_1} \mid \ldots \mid a_n(\pi_1\colon \mathcal{T})?is_{a_n} \\
&\qquad\qquad\qquad \mid alt(\pi_1\colon \mathcal{T}, \pi_2\colon \mathcal{T})?is_{alt} \mid par(\pi_1\colon \mathcal{T}, \pi_2\colon \mathcal{T})?is_{par};\\
&\textbf{sort}\ \ \mathcal{A}_{Trans} = \textbf{struct}\ a_1 \mid \cdots \mid a_n;\\
&\textbf{sort}\ \ S_{Trans} = \textbf{struct}\ sol(\pi_l\colon \mathcal{A}_{Trans}, \pi_t\colon \mathcal{T});
\end{aligned}$$

The deduction rules for this process algebra are:

$$(a_1)\frac{}{a_1.x_1 \xrightarrow{a_1} x_1} \qquad \cdots \qquad (a_n)\frac{}{a_n.x_1 \xrightarrow{a_n} x_1} \qquad \textbf{(a1)}\frac{x_1 \xrightarrow{l} y_1}{x_1 + x_2 \xrightarrow{l} y_1}$$

$$\textbf{(a2)}\frac{x_2 \xrightarrow{l} y_2}{x_1 + x_2 \xrightarrow{l} y_2} \qquad \textbf{(p1)}\frac{x_1 \xrightarrow{l} y_1}{x_1 \parallel x_2 \xrightarrow{l} y_1 \parallel x_2} \qquad \textbf{(p2)}\frac{x_2 \xrightarrow{l} y_2}{x_1 \parallel x_2 \xrightarrow{l} x_1 \parallel y_2}$$

As no conditions (other than *true*) appear in these deduction rules we do not consider them in the remainder of this section. To accommodate the (auxiliary) computation we introduce the following functions and variables:

$$\begin{array}{ll}\textbf{map} & R_{Trans}, R_{a_1}, \ldots, R_{a_n}, R_{\mathbf{a1}}, R_{\mathbf{a2}}, R_{\mathbf{p1}}, R_{\mathbf{p2}} \colon \mathcal{T} \to Set(S_{Trans});\\ & \sigma^{x_1}, \sigma^{x_2}, \sigma^{y_1}, \sigma^{y_2}, \sigma^{y_1 \| x_2}, \sigma^{x_1 \| y_2} : \mathcal{T} \to \mathbb{B};\\ & \mu^{x_1}_{x_1}, \mu^{y_1}_{y_1}, \mu^{y_2}_{y_2}, \mu^{y_1 \| x_2}_{y_1}, \mu^{y_1 \| x_2}_{x_2}, \mu^{x_1 \| y_2}_{x_1}, \mu^{x_1 \| y_2}_{y_2} : \mathcal{T} \to \mathcal{T};\\ \textbf{var} & v \colon \mathcal{T};\end{array}$$

The sort S_{Trans} refers to the declaration defined in Section 3.2. The overall relation function we define as:

$$\textbf{eqn} \quad R_{Trans}(v) = R_{a_1}(v) \cup \ldots \cup R_{a_n}(v) \cup R_{\mathbf{a1}}(v) \cup R_{\mathbf{a2}}(v) \cup R_{\mathbf{p1}}(v) \cup R_{\mathbf{p2}}(v);$$

Then the resulting equations for the action prefix terms are, for each $\alpha \in \mathcal{A}$

$$\begin{array}{lll}\textbf{eqn} & \sigma^{x_1}(v) & = \mathit{true};\\ & \mu^{x_1}_{x_1}(v) & = v;\\ & R_\alpha(v) & = \{s \colon S_{Trans} \mid is_\alpha(v) \wedge \sigma^{x_1}(\pi_t(s)) \wedge \mu^{x_1}_{x_1}(\pi_t(s)) \approx \pi_1(v)\};\end{array}$$

The required equations for deduction rule **(a1)** are:

$$\begin{array}{lll}\textbf{eqn} & \sigma^{y_1}(v) & = \mathit{true};\\ & \mu^{y_1}_{y_1}(v) & = v;\\ & R_{\mathbf{a1}}(v) & = \{s \colon S_{Trans} \mid is_{alt}(v) \wedge \ \sigma^{y_1}(\pi_t(s))\\ & & \qquad \wedge \exists_{l_1 \colon \mathcal{A}_{Trans}}(sol\left(l_1, \mu^{y_1}_{y_1}(\pi_t(s))\right) \in R_{Trans}(\pi_1(v)))\};\end{array}$$

For deduction rule **(p1)**, the following set of equations is constructed:

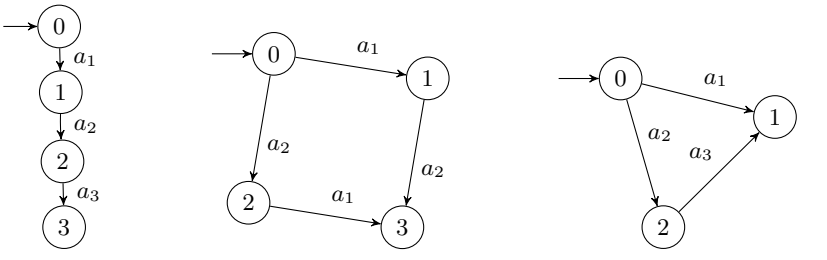
$$\begin{array}{lll}\textbf{eqn} & \sigma^{y_1 \| x_2}(v) & = is_{par}(v) \wedge \sigma^{y_1}(\pi_1(v)) \wedge \sigma^{x_2}(\pi_2(v));\\ & \sigma^{y_1}(v) & = \mathit{true};\\ & \sigma^{x_2}(v) & = \mathit{true};\\ & \mu^{y_1 \| x_2}_{y_1}(v) & = \mu^{y_1}_{y_1}(\pi_1(v));\\ & \mu^{y_1}_{y_1}(v) & = v;\\ & \mu^{y_1 \| x_2}_{x_2}(v) & = \mu^{x_2}_{x_2}(\pi_2(v));\\ & \mu^{x_2}_{x_2}(v) & = v;\\ & R_{\mathbf{p1}}(v) & = \{s \colon S_{Trans} \mid is_{par}(v) \wedge \ \sigma^{y_1 \| x_2}(\pi_t(s))\\ & & \qquad \wedge \exists_{l_1 \colon \mathcal{A}_{Trans}}(sol(l_1, \mu^{y_1 \| x_2}_{y_1}(\pi_t(s))) \in R_{Trans}(\pi_1(v))\\ & & \qquad \wedge \ \mu^{y_1 \| x_2}_{x_2}(\pi_t(s)) \approx \pi_2(v))\};\end{array}$$

The treatment of deduction rules **(a2)** and **(p2)** is analogous to the treatment of rules **(a1)** and **(p1)**.

To perform a meaningful analysis for the closed term p, we provide the following LPE, instantiated by p as:

$$\begin{array}{ll}\textbf{proc} & X(v \colon \mathcal{T}) = \sum_{s \colon S_{Trans}} s \in R_{Trans}(v) \to \pi_l(s) \cdot X(\pi_t(s));\\ \textbf{init} & X(p);\end{array}$$

To illustrate that the method is effective, Figure 1 provides graphs generated by the mCRL2 tool-set (release-March 2011), that are obtained by applying the framework. In each case, the initial process parameter p from sort $\mathcal{T}$, which generates the labeled transition system, is provided in the caption below the the graphs. The tools that have been used to generate the pictures are subsequently txt2lps and lps2lts. The first tool reads a textual LPS and stores it into the binary LPS format. The second tool unfolds an LPS into a labeled transition system.



(a) $a_1(a_2(a_3(zero)))$ (b) $par(a_1(zero), a_2(zero))$ (c) $alt(a_1(zero), a_2(a_3(zero)))$

Fig. 1. Three different specifications, as generated by the mCRL2 tool-set

5 Implementation

In order to implement a specification, we require a finite number of deduction rules and a finite signature, such that we can generate a finite textual specification. Furthermore we need to apply two restrictions, in order to conduct an analysis. The first restriction applies to the use of actions. The second restriction applies to the use of quantifiers.

In the example we use elements of sort $\mathcal{A}_{Trans}$ (part of the data specification) as actions in mCRL2. Within the mCRL2 language the direct use of data sorts as actions is prohibited. In fact, mCRL2 requires two separate declarations. To overcome this limitation, we declare a (dummy) action with a data parameter of sort $\mathcal{A}_{Trans}$ and use this data parameter to encode the SOS-action. So instead of $p \xrightarrow{a} p'$, we get $p \xrightarrow{Trans(a)} p'$, where *Trans* is the dummy action name carrying $\mathcal{A}_{Trans}$ as its parameter.

The second restriction applies to the use of quantifiers. The mCRL2 language allows the user to specify existential ($\exists$) quantifiers, but their evaluation within the tool-set is currently being developed. The existential quantification over the action labels can be dealt with by the tool-set since these concern a finite domain.

The existential quantifiers over the z_i variables are not necessarily over a finite domain. The mCRL2 tool-set cannot compute these in all cases. However, the expressions $\exists_{z_i:\mathcal{T}} sol(l_i, z_i) \in R_{Trans}(\pi_i(p))$ can be replaced by expressions $l_i \in R^l(\pi_i(p))$, where the function R_l is like R_{Trans} but instead of returning a set of solutions, which consists of labels and terms, it returns only a set of labels. Let $R^l, (R^l_d)_{d\in\mathcal{D}} \colon \mathcal{T} \to Set(\mathcal{A}_{Trans})$ be the derived function along with its auxiliary functions. Then $R^l = \bigcup_{d\in\mathcal{D}} R^l_d$, where the auxiliary functions are defined as:

$$\textbf{eqn}\quad R^l_d(p) = \{a\colon \mathcal{A}_{Trans} \mid is_f(p) \wedge \exists_{l_{i_1},\ldots,l_{i_{|I|}}\colon \mathcal{A}_{Trans}}(\, Cond_d(l_{i_1},\ldots,l_{i_{|I|}},a) \wedge \bigwedge_{i\in I} l_i \in R^l(\pi_i(p)))\};$$

6 Extension of the Framework

Although we have not shown it in this paper, we believe that there are no reasons why the mCRL2 tool-set would not be able to deal with multi-sorted signatures. The main adaptation to the presented method is that for each sort in the signature a different mCRL2 sort needs to be defined.

Also extending the framework to deal with TSSs in which multiple transition relations occur poses no problem as it requires the definition of a different function R for each transition relation in the TSS. Also, different solution sorts must be provided based on the arities of the involved transition relations. mCRL2 requires that the sorts do not share function symbol names. The example in the end of this section illustrates the treatment of multiple transition relations.

In the previous section we have only dealt with sets of actions labels that are finite and that are therefore easily captured by means of a structured sort in mCRL2. mCRL2 allows the use of much more involved sorts. As long as the label set can be captured as a sort in mCRL2 we can also deal with infinite label sets. A problem may arise in the impossibilities of the tool-set in dealing with the existential quantifiers in that case.

In the remainder of this section we discuss how predicates can be dealt with. Predicates are a useful addition to TSSs [5]. Predicates are used to express behavioral properties, like termination and divergence. A deduction rule d in a TSS with transition relation symbol $\longrightarrow$ and predicate P is of the form:

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{Px_j \mid j \in J\}}{Pf(x_1,\ldots,x_{ar(f)})}[Cond_d]$$

or

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{Px_j \mid j \in J\}}{f(x_1,\ldots,x_{ar(f)}) \xrightarrow{l} t}[Cond_d]$$

where all of $x_1,\ldots,x_{ar(f)}$ and y_i, for $i \in I$ are distinct variables, $f \in \Sigma$, $I, J \subseteq \{1,\ldots,ar(f)\}$ and $I \cap J = \emptyset$, t is a process term that only contains variables from $\{x_k \mid k \notin I \cup J\} \cup \{y_i \mid i \in I\}$ and does not have repeated occurrences of variables, l_i's and l are labels and $Cond_d$ is a condition on the labels of the premises and the conclusion (if any).

Predicates can be considered a special type of transition relation, with special transition labels. Therefore, we introduce a special transition relation symbol $\xrightarrow{P}$ for each predicate P. Then the above deduction rules can be represented by:

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{x_j \xrightarrow{P} y_j \mid j \in J\}}{f(x_1,\ldots,x_{ar(f)}) \xrightarrow{P} f(z_1,\ldots,z_{ar(f)})}[Cond_d]$$

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{x_j \xrightarrow{P} y_j \mid j \in J\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{l} t}[Cond_d]$$

where $z_k = x_k$ for all $k \notin I \cup J$, and $z_k = y_k$, otherwise. Since we assumed I and J to be disjoint, these rules are in the De Simone format.

For the new transition relation $\xrightarrow{P}$ (representing the predicate relation P) we define the sorts $\mathcal{A}_{Pred}$ with single element P, and S_{Pred} and function R_{Pred}, such that for all $s \in \mathcal{C}(\Sigma)$ and labels $P \in \mathcal{A}_{Pred}$, holds:

$$sol(P, s) \in R_{Pred}(s) \qquad \text{iff} \qquad s \xrightarrow{P} s$$

Note, that predicates modeled in this way, appear as a self-loop transition in a labeled transition system. To emphasize the difference between action transitions and predicate transitions, we use dummy action *Pred* for predicate transitions.

sort $\mathcal{A}_{Pred} = \mathbf{struct}\ P;$
sort $S_{Pred} = \mathbf{struct}\ sol(\pi_l \colon \mathcal{A}_{Pred}, \pi_t \colon \mathcal{T});$
act $Pred \colon \mathcal{A}_{Pred};$

$$\mathbf{proc}\ X(p \colon T) = \textstyle\sum_{s \colon S_{Trans}} s \in R_{Trans}(p) \rightarrow Trans(\pi_l(s)) \cdot X(\pi_t(s)) + \sum_{s \colon S_{Pred}} s \in R_{Pred}(p) \rightarrow Pred(\pi_l(s)) \cdot X(\pi_t(s));$$

Predicate application. In this example we extend MPT with termination. By introducing termination, the signature as mentioned in Section 4, is extended with the function symbol 1. Within the MPT extension it is common to write $x\downarrow$ instead of $\downarrow x$. The deduction rules for this extension are:

$$\mathbf{(t1)}\frac{}{1\downarrow} \qquad \mathbf{(t2)}\frac{x_1\downarrow}{x_1 + x_2\downarrow} \qquad \mathbf{(t3)}\frac{x_2\downarrow}{x_1 + x_2\downarrow}$$

The deduction rules that we obtain by replacing the predicates by transition relations are the following:

$$\mathbf{(t1)}\frac{}{1 \xrightarrow{\downarrow} 1} \qquad \mathbf{(t2)}\frac{x_1 \xrightarrow{\downarrow} y_1}{x_1 + x_2 \xrightarrow{\downarrow} y_1 + x_2} \qquad \mathbf{(t3)}\frac{x_2 \xrightarrow{\downarrow} y_2}{x_1 + x_2 \xrightarrow{\downarrow} x_1 + y_2}$$

Now, we first extend the signature by adding a nullary constructor function *one* representing the constant 1 and a recognizer function is_{one} as follows:

$$\mathbf{sort}\ \mathcal{T} = \mathbf{struct}\ \ zero?is_{zero} \mid one?is_{one} \mid a_1(\pi_1 \colon \mathcal{T})?is_{a_1} \mid \ldots \mid a_n(\pi_1 \colon \mathcal{T})?is_{a_n} \mid alt(\pi_1 \colon \mathcal{T}, \pi_2 \colon \mathcal{T})?is_{alt} \mid par(\pi_1 \colon \mathcal{T}, \pi_2 \colon \mathcal{T})?is_{par};$$

To compute the solution belonging to the termination predicate we introduce function the R_{Pred}, supported by three auxiliary functions $R_{\mathbf{t1}}, R_{\mathbf{t2}}, R_{\mathbf{t3}}$. Valid solutions for predicates are computed by:

$$\textbf{map}\ \ R_{Pred}, R_{\mathbf{t1}}, R_{\mathbf{t2}}, R_{\mathbf{t3}} \colon \mathcal{T} \to Set(S_{Pred});$$
$$\textbf{eqn}\ \ R_{Pred}(v) = R_{\mathbf{t1}} \cup R_{\mathbf{t2}} \cup R_{\mathbf{t3}};$$

where the auxiliary functions are defined as:

$$\begin{aligned}
\textbf{eqn}\ \ R_{\mathbf{t1}}(v) &= \{s \colon S_{Pred} \mid is_{one}(v) \wedge \pi_t(s) \approx v \wedge \pi_l(s) \approx \downarrow\};\\
R_{\mathbf{t2}}(v) &= \{s \colon S_{Pred} \mid is_{alt}(v) \wedge \sigma^{y_1+x_2}(\pi_t(s))\\
&\qquad \wedge\ sol(\downarrow, \mu^{y_1+x_2}_{y_1}(\pi_t(s))) \in R_{Pred}(\pi_1(v)) \wedge \mu^{y_1+x_2}_{x_2}(\pi_t(s)) \approx \pi_2(v)\};\\
R_{\mathbf{t3}}(v) &= \{s \colon S_{Pred} \mid is_{alt}(v) \wedge \sigma^{x_1+y_2}(\pi_t(s))\\
&\qquad \wedge\ sol(\downarrow, \mu^{x_1+y_2}_{y_2}(\pi_t(s))) \in R_{Pred}(\pi_2(v)) \wedge \mu^{x_1+y_2}_{x_1}(\pi_t(s)) \approx \pi_1(v)\};
\end{aligned}$$

To illustrate the use of predicates within the framework, consider Figure 2 that shows a generated example with the mCRL2 tool-set. The initial specification p is shown in the caption. Here the process can either perform action a_1 and deadlock or perform action a_2 and terminate successfully. The tools used are identical to those used in our previous example.

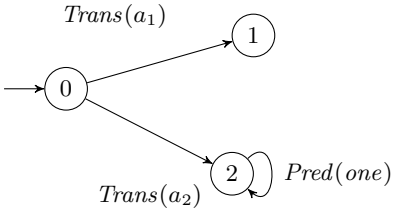


Fig. 2. Example of a predicate, generated by the mCRL2 tool-set for $alt(a_1(zero), a_2(one))$

7 Related Work

SOS meta-theory research is mainly aimed at proving useful properties about TSSs [2,32] such as congruence results [23], deriving equational theories [1], conservative extensions [21], and soundness of axioms [3]. Research on how to implement them is underexposed. Most of the related work is performed with the Maude model checker [36]. Other authors have studied the link between the rewriting logic [26] and SOS both from a theoretical [10,11,28,19,32] as well as practical point of view [12,13,19,31,39,40].

In [13], the outline of a translation from Modular SOS (MSOS) [29,30] to the Maude rewriting logic is given and proven correct. The translation is straightforward and the technical twist is in the decomposition of labels, *e.g.*, to the structure of the labels in MSOS. A more elaborate explanation of this can be found in [11]. Within the work of [40], they try to capture the CCS semantics rewrites. While rewrites have no labels, labels are encoded as the result of a rewrite rule, *e.g.*, the CCS transition of $p \xrightarrow{a} q$ is written as $a.p \longrightarrow \{a\}p$. Though this is a correct transition, $(a.p) \parallel q \longrightarrow (\{a\}p) \parallel q$ is not, since the right-hand side term is not well formed. To overcome this problem, they introduce a dummy

operator by which they extend the semantics in order to generate the transitive closure (p34-p38). Basically, rewrites can only be performed on the outermost function symbol and the result needs to be constructed as such. Since we use tuples to store a solution, rather than encoding it into a single term, we do not have to compute the transitive closure.

In [31,38,39,40] we see that the most noticeable difference is the formalism in which they express the TSS. In these works the authors stick to a representation for which hardly any tooling for formal analysis is available, or needs to be developed from scratch. This hinders a formal analysis. We have chosen a formalism, that is supported by a collection of tools that is specially aimed at performing formal analysis.

LETOS [25] is a tool environment that generates LaTeX documents and executable animations in Miranda [37]. This can be accomplished for a wide range of semantics, including some deterministic SOS forms. Since LETOS can only deal with deterministic semantics, it poses some problems when analyzing the behavior of concurrent (non-deterministic) systems.

An approach for implementing SOS rules is presented in [15], which combines (unconditional) term-rewriting and λ-calculus for simulation. It demonstrates how SOS can be used in proof tools based on term rewriting. For that the Larch Prover [24] is used, and explained in [14]. Their method aims to demonstrate and prove the equivalence between different semantics definitions. We, however, aim at creating a bridge that closes the gap between a language for specification and a language for performing analysis. Furthermore, we include conditions and predicates, whereas they only allow predicates.

Process Algebra Compiler [16] is a tool that takes the signature and the SOS rules of a language and generates a LEX/YACC scanner/parser as well as verification libraries (Lisp and in Standard ML which are respectively compiled with the kernels of the MAUTO tool [9] and the Concurrency Workbench [17]). In fact, PAC is a compiler that can be used as a front-end for verification tools. With the help of so-called back-end procedures, they generate the required routines for the different target systems, by relating concepts from the original language to those in the target formalism. How the relationship is defined between them, still needs to be addressed by the user. As our work describes such a relation, this method can be implemented into PAC.

8 Assessment and Future Work

In this paper we have demonstrated that a subclass of SOSs, namely those adhering to the De Simone format, can be transformed into a Linear Process Specification in the mCRL2 language. These can be subsequently accommodated with the mCRL2 tool-set. Although we have selected mCRL2 as our specification/implementation language, we do not foresee any difficulties when choosing another language as long as it has the same expressive power, *e.g.*, it facilitates a higher-order rewrite system to compute set comprehensions and a transition generator to (exhaustively) explore behavior.

The work presented here originates from work carried out as part of the KWR 09124 project LithoSysSL at ASML. The core activity within this project is to investigate how to formalize a language-oriented, domain specific modeling environment and use it for specification, verification and validation purposes within the Lithography domain. During the project we have performed several successful analyses, which are based on the framework as presented here. Ad-hoc concepts that have been incorporated include multiple signatures, complex predicates for transition synchronization and the enabling of transition via data. The framework has been successfully used to study the composition of language elements, which revealed unintended execution behavior [35].

Extensions towards multi-sorted transition specifications with multiple transition relations are currently considered. Also, we are looking into accommodation of extensions of the De Simone format. We claim to be able to deal with negative premises of the form $x_i \stackrel{l}{\not\rightarrow}$, with copying of variables in the premises and in the target term of the conclusion. This means that we can deal with TSSs where the deduction rules have finitely many premises and are in the GSOS rule format [7]. Besides the extension towards the GSOS format, we also have preliminary evidence that look-ahead in premises can be dealt with to some extent and that the use of state vectors in the TSS (as far as these are expressible in the mCRL2 data language) is feasible.

By incorporating these extensions, we will be able to transform more languages towards the mCRL2 tool-set and facilitate formal behavioral analysis of a wider range of (domain specific) languages. Finally, we are planning to investigate the scalability of our framework. This includes aspects such as the size of the signature, size of the syntax and the number and complexity of deduction rules.

References

1. Aceto, L., Bloom, B., Vaandrager, F.W.: Turning SOS Rules into Equations. Inf. Comput. 111(1), 1–52 (1994)
2. Aceto, L., Fokkink, W., Verhoef, C.: Conservative Extension in Structural Operational Semantics. In: Current Trends in Theoretical Computer Science, pp. 504–524 (2001)
3. Aceto, L., Ingolfsdottir, A., Mousavi, M.R., Reniers, M.A.: Algebraic Properties for Free? Bulletin of the EATCS 99, 81–104 (2009)
4. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press, Cambridge (2009)
5. Baeten, J.C.M., Verhoef, C.: A Congruence Theorem for Structured Operational Semantics with Predicates. In: CONCUR, pp. 477–492 (1993)
6. Bezem, M., Bol, R.N., Groote, J.F.: Formalizing Process Algebraic Verifications in the Calculus of Constructions. Formal Asp. Comput. 9(1), 1–48 (1997)
7. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. J. ACM 42(1), 232–268 (1995)
8. Bol, R.N., Groote, J.F.: The Meaning of Negative Premises in Transition System Specifications. J. ACM 43(5), 863–914 (1996)

9. Boudol, G., Roy, V., de Simone, R., Vergamini, D.: Process Calculi, from Theory to Practice: Verification Tools. In: Automatic Verification Methods for Finite State Systems, pp. 1–10 (1989)
10. Braga, C., Meseguer, J.: Modular Rewriting Semantics in Practice. ENTCS 117, 393–416 (2005)
11. de O. Braga, C.: Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro (2001)
12. de O.Braga, C., Haeusler, E.H., Bevilacqua, V., Mosses, P.D.: Maude action tool: Using reflection to map action semantics to rewriting logic. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, p. 407. Springer, Heidelberg (2000)
13. de O. Braga, C., Haeusler, E.H., Meseguer, J., Mosses, P.D.: Mapping Modular SOS to Rewriting Logic. In: LOPSTR, pp. 262–277 (2002)
14. Buth, K.H.: Using SOS Definitions in Term Rewriting Proofs. In: Larch, Workshops in Computing, pp. 36–54. Springer, Heidelberg (1992)
15. Buth, K.-H.: Simulation of SOS Definitions with Term Rewriting Systems. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788, pp. 150–164. Springer, Heidelberg (1994)
16. Cleaveland, R., Madelaine, E., Sims, S.: A Front-End Generator for Verification Tools. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 153–173. Springer, Heidelberg (1995)
17. Cleaveland, R., Sims, S.: The NCSU Concurrency Workbench. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 394–397. Springer, Heidelberg (1996)
18. de Simone, R.: Higher-Level Synchronising Devices in Meije-SCCS. Theor. Comput. Sci. 37, 245–267 (1985)
19. Degano, P., Gadducci, F., Priami, C.: A Causal Semantics for CCS via Rewriting Logic. Theor. Comput. Sci. 275(1-2), 259–282 (2002)
20. Fokkink, W.: Modelling Distributed Systems. Springer, Heidelberg (2007)
21. Fokkink, W., Verhoef, C.: A Conservative Look at Operational Semantics with Variable Binding. Inf. Comput. 146(1), 24–54 (1998)
22. Groote, J.F., Mathijssen, A.J.H., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: The Formal Specification Language mCRL2. In: Methods for Modelling Software Systems (MMOSS), number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
23. Groote, J.F., Vaandrager, F.W.: Structured Operational Semantics and Bisimulation as a Congruence. Inf. Comput. 100(2), 202–260 (1992)
24. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Springer-Verlag New York, Inc., New York (1993)
25. Hartel, P.H.: LETOS - a Lightweight Execution Tool for Operational Semantics. Softw., Pract. Exper. 29(15), 1379–1416 (1999)
26. Martí-Oliet, N., Meseguer, J.: Rewriting Logic as a Logical and Semantic Framework. ENTCS 4 (1996)
27. The mCRL2 toolset, `http://www.mcrl2.org/`
28. Meseguer, J.: Conditioned Rewriting Logic as a United Model of Concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
29. Mosses, P.D.: Exploiting Labels in Structural Operational Semantics. In: SAC 2004, pp. 1476–1481 (2004)
30. Mosses, P.D.: Modular Structural Operational Semantics. J. Log. Algebr. Program 61, 195–228 (2004)

31. Mousavi, M.R., Reniers, M.A.: Prototyping SOS Meta-theory in Maude. ENTCS 156(1), 135–150 (2006)
32. Mousavi, M.R., Reniers, M.A., Groote, J.F.: SOS Formats and Meta-theory: 20 Years After. Theor. Comput. Sci. 373(3), 238–272 (2007)
33. Plotkin, G.D.: A Structural Approach to Operational Semantics. J. Log. Algebr. Program 61, 17–139 (2004)
34. Stappers, F.P.M., Reniers, M.A., Weber, S.: Transforming SOS Specifications to Linear Processes. Computer Science Report No. 11-07, Eindhoven University of Technology (May 2011)
35. Stappers, F.P.M., Weber, S., Reniers, M.A., Andova, S., Nagy, I.: Formalizing a Domain Specific Language using SOS: An Industrial Case Study. In: SLE. LNCS (2011)
36. The Maude system, `http://maude.cs.uiuc.edu/`
37. Turner, D.A.: Miranda: A Non-Strict Functional Language with Polymorphic Types. In: FPCA, pp. 1–16 (1985)
38. Verdejo, A.: Building Tools for LOTOS Symbolic Semantics in Maude. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 292–307. Springer, Heidelberg (2002)
39. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. ENTCS 71 (2002)
40. Verdejo, A., Martí-Oliet, N.: Executable Structural Operational Semantics in Maude. J. Log. Algebr. Program 67(1-2), 226–293 (2006)

Formal Verification of Real-Time Data Processing of the LHC Beam Loss Monitoring System: A Case Study

Naghmeh Ghafari[1], Ramana Kumar[2], Jeff Joyce[1],
Bernd Dehning[3], and Christos Zamantzas[3]

[1] Critical Systems Labs, Vancouver, BC, Canada
[2] University of Cambridge, Cambridge, UK
[3] CERN, Geneva, Switzerland

Abstract. We describe a collaborative effort in which the HOL4 theorem prover is being used to formally verify properties of a structure within the Large Hadron Collider (LHC) machine protection system at the European Organization for Nuclear Research (CERN). This structure, known as *Successive Running Sums* (SRS), generates the primary input to the decision logic that must initiate a critical action by the LHC machine protection system in response to the detection of a dangerous level of beam particle loss. The use of mechanized logical deduction complements an intensive study of the SRS structure using simulation. We are especially interested in using logical deduction to obtain a generic result that will be applicable to variants of the SRS structure. This collaborative effort has individuals with diverse backgrounds ranging from theoretical physics to system safety. The use of a formal method has compelled the stakeholders to clarify intricate details of the SRS structure and behaviour.

1 Introduction

The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is a high-energy particle accelerator. It is designed to provide head-on collisions of protons at a center of a mass energy of 14 TeV for high-energy particle physics research. In order to reach the required magnetic field strengths, the LHC has superconducting magnets cooled with superfluid helium. Due to the high energy stored in the circulating beams (700 MJ), if even a small fraction of the beam particles deposit their energy in the equipment, they can cause the superconductors to transition to their normal conducting state. Such a transition is called a *quench*. The consequences of a quench range from several hours of downtime (for cooling the magnets down to their superconducting state), to months of repairs (in the case of equipment damage).

The main strategy for protecting the LHC is based on the Beam Loss Monitoring System (BLMS), which triggers the safe extraction of the beams if particle loss exceeds thresholds that are likely to result in a quench. At each cycle of the two counter-rotating beams around the 27 km tunnel of LHC, the BLMS records and processes several thousands of data points to decide whether the beams should be permitted to continue circulating or whether their safe extraction should be triggered. The processing includes analysis of the loss pattern over time and of the energy of the beam.

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 212–227, 2011.

The BLMS must respond to dangerous losses quickly, but determining whether losses are dangerous may require analysis of loss data recorded over a long period of time. Furthermore, the BLMS must continue recording large amounts of data in real-time while processing. To achieve these goals, the BLMS maintains approximate cumulative sums of particle losses over a variety of sizes of moving windows. The component responsible for maintaining these sums is called *Successive Running Sums* (SRS). The SRS component is implemented in hardware, in order to be fast enough to work in real-time, and on Field Programmable Gate Arrays (FPGAs) in particular so that they can be easily reprogrammed with future upgrades [16].

The SRS component has a complex structure and the correctness of its behaviour is critical for safe and productive use of the LHC. Any error in the SRS implementation would compromise either the availability of the LHC (unnecessary request for a beam dump) or its safety (not triggering a necessary beam dump). The current approach for analyzing the SRS implementation is simulation of its behavior on sample streams of input for different loss scenarios [15].

In this paper, we describe a formal verification approach, based on logical deduction using HOL4 theorem prover [8,13], to analyzing the SRS implementation. Our high level proof strategy takes advantage of the regular structure of the SRS, which consists of multiple layers of shift registers and some simple arithmetic hardware. There is a degree of regularity in how the output of each layer is used as input to the next layer. There is also a degree of regularity in the timing of each layer with respect to its position in the stack of layers. This regularity serves as a basis for inductive reasoning, which makes the amount of verification effort impervious to the number of layers in the structure.

Our interest in using formal methods was originally motivated by questions about the SRS that arose in the course of an external technical audit of the BLMS performed by two of the co-authors, Ghafari and Joyce, and their colleagues. Compared to test-based methods, like simulation, formal methods not only offer much higher confidence in the correctness of a system's behavior, but also help improve our understanding of its specification. One of the challenges in pursuing a formal verification approach for SRS was capturing the intricate details of the system's specification via experiment and refinement with a team of different backgrounds and expertise. Our confidence in the SRS design as a result of this effort ultimately rests upon our deep understanding of why the design is correct rather than the fact that we obtained "Theorem Proved" as the final output of a software tool. In particular, our use of mechanized logical deduction was a highly iterative process that incrementally refined our understanding of (1) the implementation (2) the intended behavior and (3) the "whiteboard-level" argument or explanation for why the implementation achieves the intended behaviors. The most important use of HOL4 was its role as an "implacable skeptic" that insisted we provide justification and compelled us to clarify the details [11].

Our contributions in this paper are: a formal model of the SRS component of the BLMS, a formal analysis of its behavior, and commentary on the process and outcomes of taking a formal approach. We give an overview of the BLMS in Section 2, and describe the SRS component in particular in Section 3. In Section 4, we describe our

Fig. 1. An ionization chamber installed on the side of the magnet in the tunnel

approach to formal verification, and present the formal model and results. Finally, we reflect on the process, summarising lessons learned and future directions, in Section 5.

2 BLMS Overview

The main purpose of the BLMS is to measure particle loss, and to request beam extraction if the loss level indicates that a quench is likely to occur. The physical principle underlying particle loss measurement [4,16] is the detection of energy deposited by *secondary shower particles* using specially-designed detectors called *ionization chambers* (see Figure 1). There are approximately 4000 ionization chambers strategically placed on the sides of the magnets all around the LHC tunnel underground. The ionization chambers produce electrical signals, based on the recording of shower particles, which are read out by *acquisition cards*. Acquisition cards, also located in the tunnel and therefore implemented by radiation-tolerant electronics, acquire and digitize the data and transmit the digitized data to the surface above the tunnel using optical links. At the surface, data processing cards named *BLETCs* receive the data and decide whether or not the beam should be permitted to be injected or to continue circulating. Each acquisition card receives data from eight ionization chambers, and each BLETC receives data from two acquisition cards. A BLETC provides data to the Logging, Post Mortem, and Collimation systems that drive on-line displays in the control room, perform long-term storage for offline analysis, and setup the collimators automatically. Due to demanding performance requirements, BLETCs are implemented on FPGAs, which include the resources needed to implement complex processing and can be reprogrammed making them ideal for future upgrades or system specification changes.

Figure 2 shows a block diagram of the processes on a BLETC FPGA. In the following, we briefly describe each of the four main processing blocks on a BLETC card.

(a) Receive, Check, and Compare (RCC): The RCC block receives data directly from the acquisition cards, and attempts to detect erroneous transmissions by using Cyclic Redundancy Check and 8B/10B algorithms [7,14].

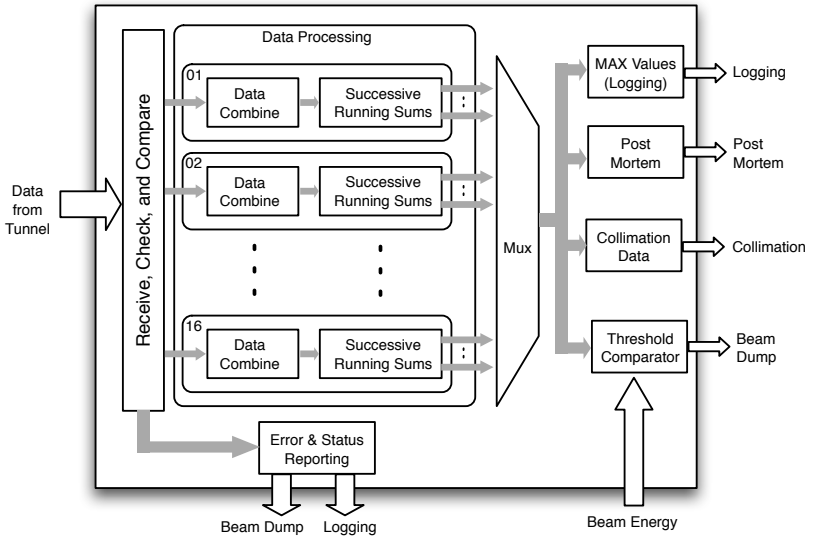


Fig. 2. Block diagram of a BLETC card

(b) Data Processing: Whether or not a quench results from particle loss depends on the loss duration and the beam energy. Given the tolerance acceptable for quench prevention, the quench threshold versus loss duration is approximated by the minimum number of sliding integration windows (called *running sums*) fulfilling the tolerance. In order to achieve the required *dynamic range* (domain of variation of losses), the detectors use both Current-to-Frequency converter and Analogue-to-Digital converter circuitries. The Data Combine block merges these two types of data coming from a detector so as to send a single value, referred to as a *count*, to the SRS block. The implementation of the SRS block is described in Section 3.

(c) Threshold Comparator: Every running sum needs to be compared to the threshold determined by the beam energy reading at that moment. The comparator initiates a beam dump request if any of the running sums is higher than its corresponding threshold. Beam dump requests are forwarded to the Beam Interlock System which initiates the beam dump. There are 12 running sums calculated for each 16-detector channel allocated to a BLETC card. There are 32 levels (0.45 to 7 TeV) of beam energy and each processing module holds data only for those 16 connected detectors. Thus, a total of 6,144 threshold values need to be held on each card.

(d) Logging, Post Mortem and Collimation: To be able to trace back the loss signal development, the BLMS stores the loss measurement data. This data is sent to the Logging and Post-Mortem systems for online viewing and storage. For the purpose of supervision, the BLMS drives an online event display to show error and status information recorded by the tunnel electronics and the RCC process as well as the maximum loss rates seen by the running sums. Each BLETC card also provides data to the Collimation system for the correct alignment and setup of the collimators.

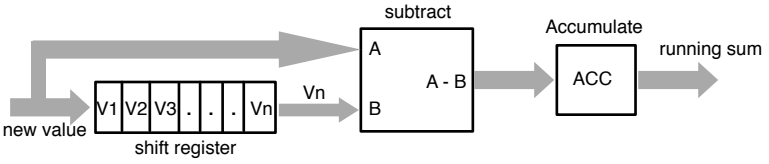


Fig. 3. Block diagram showing how to produce and maintain a continuous running sum of arriving values

3 Successive Running Sums (SRS)

Beam losses can happen at different rates, compared to the number of cycles of the beams around the tunnel. One-cycle failures are called *ultra-fast* losses. Multi-cycle losses can be classified as: *very fast* losses, which happen in less than 10 ms; *fast* losses, which happen between 10 ms and 1 s; and, *steady* losses, where the beam is lost over one second or more [12].

Processing the data collected by the detectors involves an analysis of the loss pattern over time, accounting for the energy of the beam. The processing procedure is based on the idea that a constantly updated moving window can be maintained in an accumulator by adding the incoming (newest) value and subtracting the oldest value (see Figure 3). The number of values in the window is its *integration time*. Ideally, we would have an unbounded number of windows with lengths covering the whole spectrum of times from 40 micro-seconds (the rate at which data from detectors enter a BLETC card) to 100 seconds, for detecting all losses from ultra-fast up to steady. To approximate this ideal with finite resources, the BLMS is given the tolerance acceptable for quench prevention, and the quench threshold versus loss duration curve is approximated by the minimum number of windows that meet the tolerance.

Long moving windows, that is, windows with large integration times, are required, which means keeping long histories of received count values. To accomplish this goal with relatively narrow shift registers, the SRS uses consecutive storage of sums of counts. Instead of storing all the values needed for a sum, the SRS accumulates many values as a partial sum, thereby using only a fraction of the otherwise needed memory space. The partial sums for a window with a large integration time are chosen so that they also serve as the sums calculated by a window with a smaller intergration time. This technique works by feeding the sum of one shift register's contents, every time its contents become completely updated, to the input of another shift register (see Figure 4). By cascading shift registers like this, very long moving windows can be constructed using a significantly small amount of memory. This scheme is the basis for the SRS implementation in each BLETC.

The SRS implementation minimizes resource usage by using smaller, previously calculated, running sums in the calculation of larger, later running sums, which therefore do not need extra summation values to be stored. In addition, it makes use of multipoint shift registers that are configured to give intermediate outputs, referred to as *taps*. The taps provide data outputs at certain points in the shift register chain, thus contributing to the efficient use of resources.

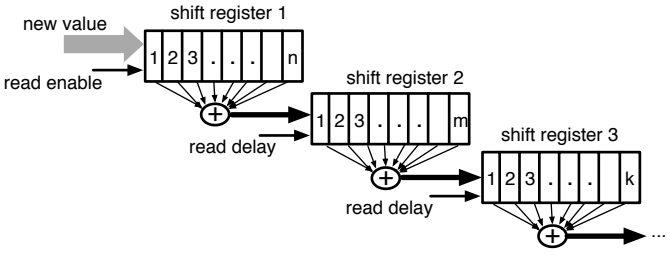


Fig. 4. Block diagram showing a configuration for efficient summation of many values

In the SRS implementation, one shift register's sum is fed as input to another shift register. Therefore, the best achievable latency of each shift register is equal to the refreshing time of its preceding shift register, i.e., the time needed to completely update its contents. The *read delay* signal (see Figure 4) of each shift register holds a delay equal to this latency to ensure correct operation. The delay is equal to the preceding shift register's delay multiplied by the number of cells to be used in the sum.

Figure 5 shows the implementation of SRS in a BLETC. It consists of 6 *slices*, where each slice computes two running sums (e.g., slice 4 computes running sums RS6 and RS7) with the use of a multipoint shift register, two subtractors and two accumulators (see Figure 6).

As shown in Table 1, cascading 6 slices is enough to reach the approximately 100 second integration limit required by the specifications given by the scientists and engineers who designed the machine protection strategy for the LHC.

4 Verification of the SRS Implementation Using HOL4

In this section, we describe our approach to formal verification of the SRS component of a BLETC, and present the formal model and results.

4.1 Introduction

Our formal verification effort uses mechanised logical deduction, or *theorem proving*. In general, theorem proving is used to show that desired properties of a system are logically implied by a formal model of the system. We use the HOL4 open source software tool [8,13], which was developed initially at the University of Cambridge, but now by an international team. HOL4 enables the construction of theories in Higher-Order Logic (HOL) [2], a formal logic with a similar expressive power to set theory that is widely used for formalising hardware and software models and statements about them. The implementation of HOL4 uses Milner's LCF approach [6]: a small "kernel" implementing the primitive rules of the logic, and convenient derived rules and tactics implemented in terms of the kernel. Every theorem ultimately comes from the kernel, and this fact provides high assurance of the logical soundness of the verification results obtained using the system.

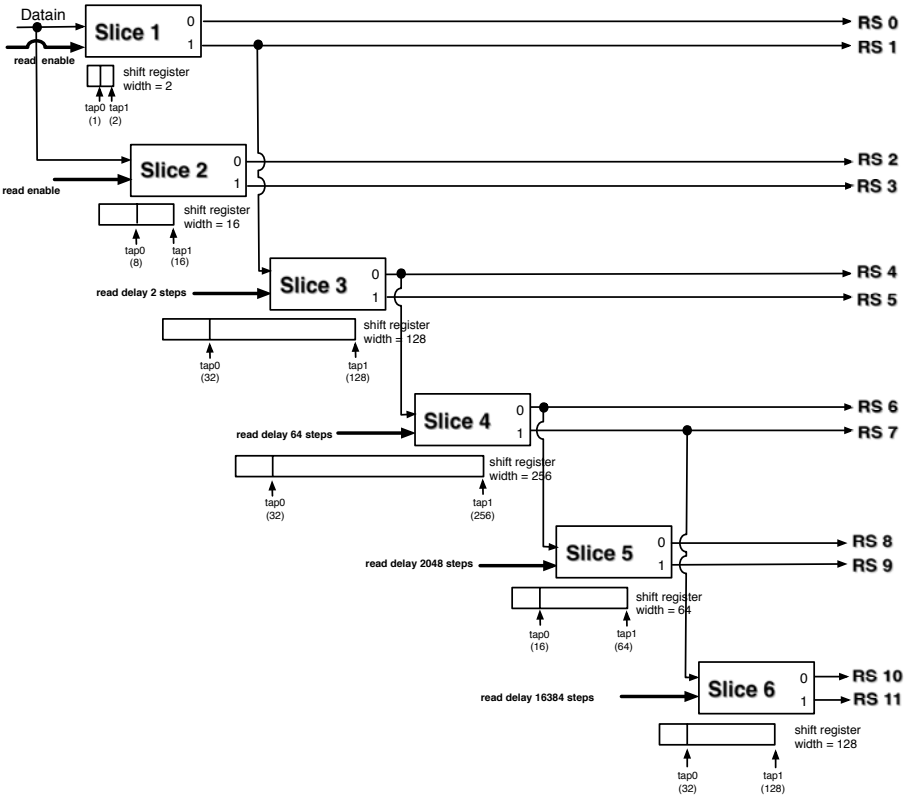


Fig. 5. Block diagram showing the implementation of SRS in a BLETC

HOL4 is an *interactive* theorem prover: the user provides the high level proof strategy by composing functions that automate common chains of logical deduction. The work described here could have been done using other systems such as HOL Light [5], Isabelle/HOL [10], ProofPower [1], PVS [9] or Coq [3]. The first three use essentially the same higher-order logic as HOL4, whilst PVS and Coq support more powerful logics. While offering less "push-button" automation than other kinds of formal verification such as model-checking, machine-assisted theorem proving using HOL4 is appropriate for verifying the SRS, since it gives a way to very explicitly parameterize the model.

Our goal is to build a generic model of the SRS structure, and to prove that it satisfies its specification, that it calculates approximate running sums of received count values within acceptable error margins. Let $\mathsf{RS}\ n$ denote, as in Figure 5, output n of the SRS structure, which is supposed to compute a sum of received count values, and let $\mathsf{true\_sum}\ n$ denote this sum. The multi-layered structure of the SRS and the read delay of each shift register result in the outputs being delayed from the $\mathsf{true\_sum}$ values. A sketch of the desired correctness statement is:

$$\forall n \cdot \mathsf{RS}\ n = \mathsf{true\_sum}\ n \pm \text{acceptable error}$$

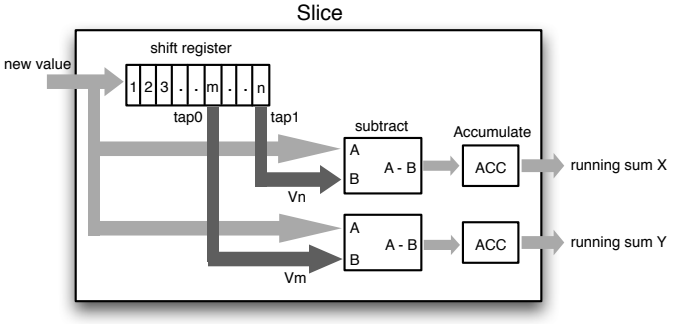


Fig. 6. Block diagram showing the implementation of each slice

Table 1. SRS configuration in BLETC

Range		Refreshing			
40 µs steps	ms	40 µs steps	ms	slice	running sum
1	0.04	1	0.04	slice 1	RS0
2	0.08	1	0.04	slice 1	RS1
8	0.32	1	0.04	slice 2	RS2
16	0.64	1	0.04	slice 2	RS3
64	2.56	2	0.08	slice 3	RS4
256	10.24	2	0.08	slice 3	RS5
2048	81.92	64	2.56	slice 4	RS6
16384	655.36	64	2.56	slice 4	RS7
32768	1310.72	2048	81.92	slice 5	RS8
131072	5242.88	2048	81.92	slice 5	RS9
524288	2097.52	16384	655.36	slice 6	RS10
2097152	83886.08	16384	655.36	slice 6	RS11

Although Figure 5 suggests that the SRS structure has only twelve outputs (i.e., $0 \leq n \leq 11$), we obtain a more generic result (that is useful in future upgrades of the system) by interpreting and proving the statement above for all values of n.

To make the above correctness statement more precise, we need to include the notion of time. The count values arrive at the input of the SRS block every 40 micro-seconds, which we abstract as a single time step in our logical model. We formalize the input stream as a function of time: $D\ t$ denotes the input value to the SRS structure at time t. In addition, the terms in the above statement depend on this stream of input counts. With these refinements, the correctness statement becomes:

$$\forall D\, n\, t \cdot \mathsf{RS}\ D\ n\ t = \mathsf{true\_sum}\ D\ n\ t \pm \text{acceptable error}$$

Prior to defining a formal model of the SRS and proving theorems about it, we developed an informal "whiteboard-level" argument for why the SRS implements its

intended behavior. The essence of this argument relies on four facts which can be established from the structure of the SRS and some details about the timing relationship between layers:

1. The shift register of each slice (except the first) is updated once the shift register of its previous[1] slice is completely updated, that is, when a count value has propagated down the full length of the previous slice's shift register.
2. The integration window of a given shift register in a slice (except the first slice) can be decomposed into a sequence of non-overlapping segments, $S_1, S_2, \ldots, S_w$ (where w is the width of the shift register) each of which is equal to the size of the integration window of the shift register of the previous slice.
3. After a period of initialization, the values stored in the shift register of a given slice (except the first) are the w outputs of the previous slice, where w is the width of the shift register of this slice.
4. After a period of initialization, the output of each slice is always equal to the sum of the contents of its shift register.

Using Facts 1, 2 and 3 in an inductive argument, we show that each cell of the shift register of a given slice, after a period of initialization, always contains the sum of the SRS inputs for one of the non-overlapping consecutive segments that make up the integration window of this slice. Then using Fact 4 and arithmetic reasoning, we can show that the output of this slice, after a period of initialization, contains the sum of the SRS inputs over its integration window.

While the above paragraph gives the appearance of a straightforward argument for the correctness of the SRS (corresponding roughly to Theorem 1 in Section 4.3), in fact the argument involves consideration of many details that arise from the formal model of the SRS presented in the next section. Using a theorem proving tool enables us to keep track of these details without losing sight of the overall goal.

4.2 Formal Model of SRS

The first step to prove the correctness statement is to build a logical model of the SRS structure. We model each building block of the SRS structure that holds a count value – for example, each cell in each shift register – as a function in HOL[2].

Our model is both a simplification and a generalization of the actual structure of SRS in BLMS in the following sense. We model an unbounded number of slices (rather than six slices), each with an unbounded number of shift register cells and taps (rather than fixed width shift registers and only two taps), simply by letting indices range over the natural numbers without explicitly giving limits. This parameterization makes the model more likely to be applicable to future versions of the system, but also fits more naturally into HOL than would a bounded model. We defined our formal model to be at a level of abstraction above the details of circuity that implements basic arithmetic operations. We use natural numbers throughout, rather than, for example, finite words

[1] Here, the phrase "previous slice" refers to the slice whose output is used as input to this slice.

[2] The acronym HOL refers to higher-order logic rather than the software tool HOL4. We use the tool to define a function in the logic.

Table 2. Descriptions of the HOL functions comprising our model of the SRS structure

Function	Intended meaning
tap $n\ x$	The position of tap x of slice n. (The first position is 0.)
input n	A pair (n', x) indicating that the input to slice n is output x of slice n'.
delay n	The number of time steps between updates of slice n.
source $D\ n\ m\ t$	The value of the cell that is the direct input to cell m of slice n, at time t, given input stream D.
SR $D\ n\ m\ t$	The value of cell m of slice n, at time t, given input stream D.
output $D\ n\ x\ t$	The output at tap x of slice n, at time t, given input stream D.
RS $D\ n\ t$	The value of running sum n at time t, given input stream D.
update_time $n\ t$	A boolean indicating whether t is an update time for slice n.

that would more accurately model count values. This level of abstraction is sufficient to answer the questions that originally motivated this work. A separate verification effort can focus on showing that a more realistic model of hardware circuitry accurately implements natural number arithmetic.

Table 2 lists the HOL functions comprising our model along with their intended meanings. The arrangement of the slices is described by functions input n and tap $n\ x$. The read delay of a slice is modeled by delay n. Each slice is modeled by three functions: SR $D\ n\ m\ t$ represents the value of each cell of the slice's shift register, source $D\ n\ m\ t$ represents the direct input of each cell, and output $D\ n\ x\ t$ represents the value of the slice's output at its taps. The function RS $D\ n\ t$ models the running sums and update_time $n\ t$ checks if it is time to refresh the contents of a slice. We have both RS and output functions, though RS is easily defined in terms of output, because RS represents an SRS output (indexed by a single number), whereas output represents an individual slice output (indexed by slice and tap numbers). By separating RS and output, we allow for designs where some slice outputs are not SRS outputs, but may still be used internally as inputs to other slices.

The formal definitions of the functions listed in Table 2 are given in Figure 7. The definition of input when $n = 0$ or when $n > 6$ does not change the structure represented, since slice 0 is a virtual slice and the real SRS has only six slices, so we give definitions convenient for theorem proving. A similar comment applies to other definitions when n, representing a slice number, is 0, or when x, representing a shift register position, is greater than 1. We define all excess taps (where $x > 1$) to be in the same position as the last tap.

As explained in Section 3, the delay of each slice is equal to the delay of the slice it receives its input from multiplied by the number of elements in the input slice used for the sum. For example in Figure 5, delay 4 = delay $3 \times$ tap $3\ 0 = 2 \times 32$. The definition of the source function states that the source of each cell m in a shift register is cell $m - 1$, except for the first cell whose source is the output of the input slice. For example, source $D\ 4\ 7\ t$ = SR $D\ 4\ 6\ t$ and source $D\ 4\ 0\ t$ = output $D\ 3\ 0\ t$. The output of each slice is computed every time the contents of its shift register are updated by adding the incoming newest value (specified by source) and subtracting its oldest value, that is the value in the cell at the tap position. The content of each cell of a shift register, SR $D\ n\ m\ t$ is also computed at every update time based on the

value of its source. Every definition in Figure 7 is *local* (only represents a small part of the SRS structure) and therefore verification against its intended meaning is relatively straightforward.

Figure 8 shows the definition of a set of additional functions required to prove our main results. The function $\mathsf{delay\_sum}\ n$ represents the cumulative delay of the preceding shift registers of a slice. For example, $\mathsf{delay\_sum}\ 4 = \mathsf{delay}\ 3 + \mathsf{delay}\ 1 + \mathsf{delay}\ 0 = 2 + 1 + 1 = 4$. The function $\mathsf{last\_update}\ n\ t$ returns the latest time not after t at which slice n updates, and $\mathsf{exact}\ D\ n\ x\ t$ computes the exact sum of consecutive input counts, without delay, that $\mathsf{output}\ D\ n\ x\ t$ is supposed to approximate.

4.3 Theorems about the Model

Our central result equates the output of a slice to a sum of consecutive input counts. More precisely, it says that if slice n was just updated, then the output at tap x is equal to the sum of $((\mathsf{tap}\ n\ x)+1)\times(\mathsf{delay}\ n)$ input values, starting from the input $\mathsf{delay\_sum}\ n$ time steps ago.

Theorem 1. *For all values of D, n, and x, the* output *of slice $n > 0$ at an update time t satisfies*

$$\mathsf{output}\ D\ n\ x\ t = \sum_{m=0}^{((\mathsf{tap}\ n\ x)+1)\times(\mathsf{delay}\ n)-1} \begin{cases} 0 & t < m + \mathsf{delay\_sum}\ n \\ D\ (t - m - \mathsf{delay\_sum}\ n) & \textit{otherwise} \end{cases}$$

Proof. The shift register of slice n is updated every $\mathsf{delay}\ n$ time steps. When updated, the values in its cells are shifted one cell. Therefore[3],

$$\mathsf{SR}\ D\ n\ (m+1)\ t = \mathsf{SR}\ D\ n\ m\ (t - \mathsf{delay}\ n)$$

By induction on m, using the above,

$$\mathsf{SR}\ D\ n\ m\ t = \mathsf{SR}\ D\ n\ 0\ (t - (m \times \mathsf{delay}\ n))$$

By induction on t, we can show that $\mathsf{output}\ D\ n\ x\ t$ is a sum of values of consecutive shift register cells,

$$\mathsf{output}\ D\ n\ x\ t = \sum_{m=0}^{\mathsf{tap}\ n\ x} \mathsf{SR}\ D\ n\ m\ t$$

Combining the last two results, $\mathsf{output}\ D\ n\ x\ t$ is a sum of consecutive values of the first shift register cell, $\mathsf{SR}\ D\ n\ 0$. Thus, we can express $\mathsf{output}\ D\ n\ x\ t$ in terms of $\mathsf{output}\ D\ n'\ x'\ t$, where $(n', x') = \mathsf{input}\ n$, since the source of $\mathsf{SR}\ D\ n\ 0$ is the previous (input) slice's output. Finally, by induction on n, we can express $\mathsf{output}\ D\ n\ x\ t$ as a function of D alone, as required, using the result above for the inductive case. □

[3] For presentational convenience, here, we do not provide details for the case when t is small (e.g., when $t < m \times \mathsf{delay}\ n$). For complete theorem statements and the proof script, please refer to `https://github.com/xrchz/CERN-LHC-BLMTC-SRS/blob/master/hol/srsScript.sml`.

$$
\begin{array}{ll}
\mathsf{tap}\ 0\ 0 = 0 & \mathsf{tap}\ 0\ x = 0 \\
\mathsf{tap}\ 1\ 0 = 1 - 1 & \mathsf{tap}\ 1\ x = 2 - 1 \\
\mathsf{tap}\ 2\ 0 = 8 - 1 & \mathsf{tap}\ 2\ x = 16 - 1 \\
\mathsf{tap}\ 3\ 0 = 32 - 1 & \mathsf{tap}\ 3\ x = 128 - 1 \\
\mathsf{tap}\ 4\ 0 = 32 - 1 & \mathsf{tap}\ 4\ x = 256 - 1 \\
\mathsf{tap}\ 5\ 0 = 16 - 1 & \mathsf{tap}\ 5\ x = 64 - 1 \\
\mathsf{tap}\ 6\ 0 = 32 - 1 & \mathsf{tap}\ 6\ x = 128 - 1 \quad \text{(where each } x > 0\text{)} \\
\ldots & \mathsf{tap}\ n\ x = 0 \quad \text{(where } x \geq 0 \text{ and } n > 6\text{)}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{input}\ 0 = (0,0) & \mathsf{input}\ 1 = (0,0) \\
\mathsf{input}\ 2 = (0,0) & \mathsf{input}\ 3 = (1,1) \\
\mathsf{input}\ 4 = (3,0) & \mathsf{input}\ 5 = (4,0) \\
\mathsf{input}\ 6 = (4,1) & \mathsf{input}\ n = (n-1,0) \quad \text{(where } n > 6\text{)}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{delay}\ 0 = 1 \\
\mathsf{delay}\ n = \mathsf{delay}\ n' \times ((\mathsf{tap}\ n'\ x) + 1) \\
\qquad\qquad \text{where } (n', x) = \mathsf{input}\ \mathsf{n} \text{ (and } n > 0\text{)}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{source}\ D\ n\ 0\ t = \mathsf{output}\ D\ n'\ x\ t \quad \text{where } (n', x) = \mathsf{input}\ \mathsf{n} \\
\mathsf{source}\ D\ n\ m\ t = \ \mathsf{SR}\ D\ n\ (m-1)\ t \quad \text{(where } m > 0\text{)}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{SR}\ D\ n\ m\ 0 = 0 \\
\mathsf{SR}\ D\ n\ m\ t = \text{if } \mathsf{update\_time}\ n\ t \text{ then} \\
\qquad\qquad\qquad \mathsf{source}\ D\ n\ m\ (t-1) \\
\qquad\qquad \text{else } \mathsf{SR}\ D\ n\ m\ t \qquad \text{(where } t > 0\text{)}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{output}\ D\ 0\ x\ t = D\ t \\
\mathsf{output}\ D\ n\ x\ 0 = 0 \\
\mathsf{output}\ D\ n\ x\ t = \text{if } \mathsf{update\_time}\ n\ t \text{ then} \\
\qquad ((\mathsf{output}\ D\ n\ x\ (t-1)) + (\mathsf{source}\ D\ n\ 0\ (t-1))) - (\mathsf{SR}\ D\ n\ (\mathsf{tap}\ n\ x)\ (t-1)) \\
\qquad\qquad \text{else } \mathsf{output}\ D\ n\ x\ (t-1) \qquad \text{(where } n, t > 0\text{)}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{RS}\ D\ n\ t = \mathsf{output}\ D\ \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)\ (n \bmod 2) \\
\mathsf{update\_time}\ n\ t \iff (t \bmod \mathsf{delay}\ n = 0)
\end{array}
$$

Fig. 7. Definitions of the HOL functions comprising our model of the SRS structure. tap and input are defined to match Figure 5. Slice 0 is a virtual slice representing the SRS input; this enables a succinct definition of source.

$$
\begin{array}{l}
\mathsf{delay\_sum}\ 0 = 0 \\
\mathsf{delay\_sum}\ n = \mathsf{delay}\ n' + \mathsf{delay\_sum}\ n' \quad \text{where } (n', \_) = \mathsf{input}\ n
\end{array}
$$

$$
\begin{array}{l}
\mathsf{last\_update}\ n\ 0 = 0 \\
\mathsf{last\_update}\ n\ t = \text{if } \mathsf{update\_time}\ n\ t \text{ then } t \\
\qquad\qquad\qquad \text{else } \mathsf{last\_update}\ n\ (t-1) \quad \text{(where } t > 0\text{)}
\end{array}
$$

$$
\mathsf{exact}\ D\ n\ x\ t = \sum_{m=0}^{((\mathsf{tap}\ n\ x)+1)\times(\mathsf{delay}\ n)} \begin{cases} 0 & t < m+1 \\ D\ (t-m-1) & \text{otherwise} \end{cases}
$$

Fig. 8. Definitions of auxiliary HOL functions used while proving theorems about the model of the SRS structure

Since the outputs of a slice stay constant between update times for the slice, Theorem 1 suffices to characterize all outputs at all times. Thus, the SRS structure's outputs are equal to the exact running sums of the input counts over windows whose sizes depend on the width of the slice's shift register and position of the taps. However, the sums are delayed by the total delay across all previous slices (represented by delay_sum).

The fact that the SRS calculates exact, but delayed, sums, is captured in the next theorem, which relates output $D\ n\ x\ t$ to exact $D\ n\ x$ at an earlier time.

Theorem 2. *For all values of D, n, and x, the* output *of slice $n > 0$ at all times t satisfies*

$$\mathsf{output}\ D\ n\ x\ t = \begin{cases} 0 & \mathsf{last\_update}\ n\ t + 1 < \mathsf{delay\_sum}\ n \\ \mathsf{exact}\ D\ n\ x\ (\mathsf{last\_update}\ n\ t + 1 - \mathsf{delay\_sum}\ n) & \textit{otherwise} \end{cases}$$

Proof. The proof follows from Theorem 1, using the definition of exact, and the fact that output $D\ n\ x\ t$ has the same value as at the last update time. □

The function true_sum is easily defined in terms of exact, namely, $\mathsf{true\_sum}\ D\ n\ t = \mathsf{exact}\ D\ \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)\ (n \bmod 2)\ t$. Since RS is similarly defined in terms of the output, Theorem 2 establishes a relationship between the values of RS and of true_sum.

We call the difference between the RS values and their corresponding true_sums, caused by the delay, the *error*. To meet the tolerance acceptable for quench prevention, the SRS specification requires a bound on this error. Without restricting the input stream, the error is unbounded[4]. However, according to extensive experimental analysis, the beam loss over time follows some patterns. By formulating some characterization of the input stream as constraints on D, we can recover a bound on the error. One of these constraints is a maximum value for the difference between two consecutive input count values. This "maximum jump size" can be inferred from the highest quench level thresholds.

Under such a constraint, we have proved the follwoing result bounding the error of output $D\ n\ x\ t$ compared to exact $D\ n\ x\ t$.

Theorem 3. *For all k and D satisfying $|D\ (t' + 1) - D\ t'| \leq k$ at all times, the following holds for all slices $n > 0$, tap positions x, and times t:*

$$t > (\mathsf{tap}\ n\ x + 1) \times (\mathsf{delay}\ n) + \mathsf{delay}\ n + \mathsf{delay\_sum}\ n \implies$$
$$|\mathsf{output}\ D\ n\ x\ t - \mathsf{exact}\ D\ n\ x\ t| \leq (\mathsf{tap}\ n\ x + 1) \times (\mathsf{delay\ n}) \times k \times$$
$$(\mathsf{delay\_sum}\ n - 1 + t \bmod \mathsf{delay}\ n)$$

Furthermore, for all k there exists an input stream D_k satisfying $|D_k\ (t'+1) - D_k\ t'| \leq k$ for all t' and

$$|\mathsf{output}\ D_k\ n\ x\ t - \mathsf{exact}\ D_k\ n\ x\ t| = (\mathsf{tap}\ n\ x + 1) \times (\mathsf{delay\ n}) \times k \times$$
$$(\mathsf{delay\_sum}\ n - 1 + t \bmod \mathsf{delay}\ n)$$

for all $n > 0$, x, and $t > (\mathsf{tap}\ n\ x + 1) \times (\mathsf{delay}\ n) + \mathsf{delay}\ n + \mathsf{delay\_sum}\ n$.

[4] For any bound, we can construct an input stream that causes the bound to be exceeded by, for example, having a sequence of zeroes followed by a sequence of count values much higher than the bound.

Proof. By applying Theorem 2, we are left with an inequation involving the function exact specifically between exact $D\ n\ x\ t$ and exact $D\ n\ x$ (last_update $n\ t + 1 -$ delay_sum n). Our assumption on t ensures that we are never in the 0 case of either Theorem 2 or of the definition of exact. The function exact is defined as a sum of consecutive values of D; we need to bound the difference between one such sum and another earlier one of the same size. But if at each step the maximum difference is k, then the total difference is at most k times the distance between the ends of the two sums, as required. (We use the fact that last_update $n\ t = t - t \bmod$ delay n.) The input stream achieving this bound is given by $D_k\ t = k \times t$. □

Theorem 3 gives a bound on the error of a running sum at a given time in terms of the time step, the slice and tap numbers, and the assumed bound on the difference between consecutive counts. This bound is tight for the conditions we assume, namely that the time step is sufficiently high and that the difference between consecutive counts is bounded, in the sense that it is achievable by an input stream satisfying those conditions. To determine whether an error bound is acceptable with respect to the specification of SRS, however, it turns out to be more useful to know the relative size of the error as a fraction of the true sum. According to the specification of SRS, the running sums should have a maximum 20% relative error (($|\mathsf{RS}\ D\ n\ t - \mathsf{true\_sum}\ D\ n\ t| < 20\% \times (\mathsf{true\_sum})$). We have not yet characterized the relationship between our error bound at a given time and the true sum at that time.

The proof of Theorem 3 is straightforward when summarized, as above. However, as for all of our theorems, there are several non-trivial details underlying the high-level summary provided. For example, *proving* that the maximum difference between the end terms in a sum is the maximum difference between consecutive terms multiplied by the number of terms requires an inductive argument. Our HOL4 proof script for verification of SRS consists of 750 lines of code. We proved 69 theorems, including 14 definitions. In addition, we had to prove approximately 30 generic theorems that were added to HOL4 libraries during this work.

5 Conclusions and Lessons Learned

We have described a case study in which we used HOL4 theorem prover to verify properties of the SRS structure within the LHC machine protection system. In this case study, we built a parameterized model of the SRS structure and showed that its behavior is correct with respect to its specification. It is likely that the configuration of the slices and shift registers will need to change in the future as the understanding of the LHC and its possible weaknesses increases to accommodate more targeted protections. Thus, the parameterization in our model is crucial to make it applicable to the future upgrades.

One of our main challenges in this effort was building the formal model and formulating the correctness statements. There are three different sources of complexity that are inherent in understanding why the SRS structure implements its intended behavior: (a) the structure of the SRS: although it features considerable regularity, it includes exceptions to this regularity that complicate reasoning, e.g., the input of one slice may not be the output of the immediately previous slice, but may be the first or second output of any earlier slice or even the global input; (b) the non-trivial timing relationships

between different elements of the SRS; e.g., the frequency of updates to the contents of a shift register depends on the position of the shift register in the layered structure of the SRS; (c) arithmetic relationships that are not always intuitively apparent at first glance. While each of these sources of complexity is manageable on its own, reasoning about the correctness of the SRS is a matter of grappling with all three sources of complexity at once.

To understand the structure of the SRS, we started with a model of a simplified structure, which had a more regular arrangement of the slices and ignored the middle taps, and proceeded to a model which is closer to the SRS implementation in BLMS. In addition, we used a basic spreadsheet simulation model for a few slices for sanity testing the correctness formulae. However, some sanity tests related to Theorem 1 led us astray when we did not know the correct formula for that theorem. The formulae we conjectured worked for small values of n, and simulating large values of n was expensive, but the counterexamples were only to be found at large values of n.

The use of mechanized theorem proving to verify the correctness of the SRS behavior complements an intensive verification effort based on simulation performed at CERN. This effort targeted the validity of the SRS as an accurate and fast enough method by analyzing its behavior (a) in the boundaries of the threshold limits and (b) in expected types of beam losses, e.g., fast and steady losses. Its results showed that the current implementation of the SRS, given in Figure 5, satisfies the specification, that is, the running sums have a maximum 20% relative error.

By contrast, the results in this paper apply to all possible input streams, not just the sample inputs considered at CERN – our main contribution, as stated in Theorem 2, showed that the behavior of the SRS is correct for all possible input streams. However, we do not know if the constraint on input stream given in Theorem 3 is sufficient to satisfy the maximum 20% relative error bound. One of the potential reasons for this problem is not knowing how to characterize the input stream to the SRS. Due to the physical nature of this problem, such a characterization of the input stream is not trivial. While testing and simulation of the SRS on a limited set of input streams offers a level of confidence that the behavior of SRS satisfies this particular specification, constructing a formal proof for all possible behaviors requires a better understanding of the characteristics of the input stream. One future research direction is to investigate whether constructing such a formal proof is feasible for the SRS component of BLMS.

Another future research direction is to refine our formal model to a lower level of abstraction, closer to the hardware level (e.g., by representing count values as finite words instead of natural numbers). We are also considering verifying a model extracted from the Hardware Description Language (HDL) used to synthesize the BLETC FPGA.

Acknowledgment. The authors thank Mike Gordon at the University of Cambridge for his insightful and helpful feedback during the course of this work.

References

1. Arthan, R.: ProofPower manuals (2004), `http://lemma-one.com/ProofPower/index/index.html`
2. Church, A.: A Formulation of the Simple Theory of Types. J. Symb. Log. 5(2), 56–68 (1940)

3. Coquand, T., Huet, G.: Coq manuals (2010), http://coq.inria.fr
4. Dehning, B.: Beam loss monitoring system for machine protection. In: Proceedings of DIPAC, pp. 117–121 (2005)
5. Harrison, J.: HOL Light manuals (2010), http://www.cl.cam.ac.uk/~jrh13/hol-light
6. Milner, R.: Logic for Computable Functions: Description of a Machine Implementation. Technical report, Stanford, CA, USA (1972)
7. Nair, R., Ryan, G., Farzaneh, F.: A Symbol Based Algorithm for Hardware Implementation of Cyclic Redundancy Check (CRC). VHDL International User's Forum 0, 82 (1997)
8. Norrish, M., Slind, K.: HOL4 manuals (1998), http://hol.sourceforge.net
9. Owre, S., Shankar, N., Rushby, J., Stringer-Calvert, D.: PVS manuals (2010), http://pvs.csl.sri.com
10. Paulson, L., Nipkow, T., Wenzel, M.: Isabelle manuals (2009), http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html
11. Rushby, J.: Formal Methods and the Certification of Critical systems. CSL Technical Report 93-7, SRI International (December 1993)
12. Schmidt, R., Assmann, R.W., Burkhardt, H., Carlier, E., Dehning, B., Goddard, B., Jeanneret, J.B., Kain, V., Puccio, B., Wenninger, J.: Beam Loss Scenarios and Strategies for Machine Protection at the LHC. In: Proceedings of HALO, pp. 184–187 (2003)
13. Slind, K., Norrish, M.: A Brief Overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
14. Widmer, A.X., Franaszek, P.A.: A DC-balanced, partitioned-block, 8B/10B transmission code. IBM J. Res. Dev. 27, 440–451 (1983)
15. Zamantzas, C.: The Real-Time Data Analysis and Decision System for Particle Flux Detection in the LHC Accelerator at CERN. Ph.D. Thesis, Brunel University (2006)
16. Zamantzas, C., Dehning, B., Effinger, E., Emery, J., Ferioli, G.: An FPGA Based Implementation for Real-Time Processing of the LHC Beam Loss Monitoring System's Data. In: IEEE Nuclear Science Symposium Conference Record, pp. 950–954 (2006)

Hierarchical Modeling and Formal Verification. An Industrial Case Study Using Reo and Vereofy

Joachim Klein[1], Sascha Klüppelholz[1], Andries Stam[2], and Christel Baier[1]

[1] Technische Universität Dresden, Germany*
[2] Almende, The Netherlands

Abstract. In traditional approaches to software development, modeling precedes programming activities. Hence, models represent the intended structure and behavior of the system-to-be. The reverse case, however, is often found in practice: using models to *gain insight into an existing software system*, enabling the evolution and refactoring of the system to new needs. We report on a case study with the ASK communication platform, an existing distributed software system with multithreaded components. For the modeling of the ASK system we followed a hierarchical top-down approach that allows a high-level description of the system behavior on different levels of abstraction by applying an iterative refinement procedure. The system model is refined by decomposing the components into sub-components together with the "glue code" that orchestrates their interactions. Our model of the ASK system is based on the exogenous coordination language Reo for specifying the glue code and an automata-based formalism for specifying the component interfaces. This approach is supported by the modeling framework of the tool-set Vereofy which is used to establish several properties of the components and the coordination mechanism of the ASK system. Besides demonstrating how modeling and verification can be used in combination to gain insight into legacy software, this case study also illustrates the applicability of exogenous coordination languages such as Reo for modeling and tool-sets such as Vereofy for the formal analysis of industrial systems.

1 Introduction

In the traditional process of software development, abstract models of software are commonly used to represent the intended structure and behavior of a system-to-be. The rigorous use of formal modeling languages and verification techniques provides a smart approach for the compositional design by stepwise refinement and yields systems that are correct by construction. In practice, however, this ideal policy for the system design is not taken and complex systems are often built by linking software components that rely on program code where no formal models are available. In those cases, especially when software needs to be adapted to new needs (by others), formal models are highly desirable to obtain valuable insights for the redesign.

* The authors are supported by the EU-project Credo and the DFG-project Syanco.

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 228–243, 2011.

Coordination and interaction modeling languages [13,11], and tools to verify properties of the models such as [2,10,12], allow to gain deeper insights into the complexity of the *coordination* and *interaction* inside a software system. In this article, we illustrate that coordination languages and corresponding model checking techniques offer an elegant framework for recovering the interaction structure of a given software system by means of an industrial case study. We consider the ASK communication platform [5] (see Sec. 2) and use the exogenous coordination language Reo [4] and the model checker Vereofy [6,7,9]. Reo is a channel-based coordination language, where the glue code that orchestrates the interactions of components is compositionally built by plugging channels together, resulting in a Reo network of channels. Its library of basic channels as well as the possibility to provide customized channels and connectors allows for a wide variety of coordination patterns. Reo allows to declare the structure of a complex system by means of channels and the interfaces of components on each level of abstraction, without having to provide the operational behavior for the components or for the Reo network explicitly. Only on the lowest level of abstraction, constraint automata [8] provide the operational semantics for the most basic components and channels. The operational behavior of the entire system is implicitly given by operational semantics of the Reo network and the automata specification of the basic components. Other modeling languages like Promela (SPIN) [12] or reactive modules [3] hardly support hierarchical top-down modeling and in contrast to other formalisms like classical process algebras that provide this support, the declarative nature of Reo promotes the specification of the coordination glue code in a very intuitive and flexible manner external (exogenous) to the components. As the components remain oblivious and independent of the context in which they are used, a clear separation between coordination and computation is achieved. This facilitates iterative refinement of the components and the glue code as well as reuse of components which have been formally verified. The Reo and constraint automata approach is supported by Vereofy, a modeling and symbolic formal verification tool-set for checking operational correctness of component-based systems. Vereofy supports a hybrid approach to modeling, with the instantiation and composition of channels, connectors, and components into a Reo network handled by Vereofy's RSL (Reo Scripting Language), while the guarded command language CARML (Constraint Automata Reactive Module Language) serves to formalize the behavioral interfaces of components. The model checking engine of Vereofy supports, among others, linear and branching-time logics adapted to the Reo and constraint automata framework augmented with regular expressions for reasoning about the data flow at I/O-ports [7].

We use the ASK system to illustrate how the exogenous modeling approach based on Reo and constraint automata can be applied to model coordination mechanisms in a hierarchical way, i.e., by identifying components and component connectors, modeling them as constraint automata and Reo networks (formally specified by CARML and/or RSL code) and refining the components for lower abstraction levels. The refinement steps rely on a decomposition of components into subcomponents and their glue code, which again are formally represented by

constraint automata and/or Reo networks. We use the model checking engine of Vereofy to establish several properties of the components and the coordination mechanisms within the ASK system. Besides demonstrating how modeling and verification can be used in combination to gain insight into legacy software, this case study primarily illustrates the applicability of an exogenous modeling approach, namely Reo and the Vereofy tool-set, for the modeling and formal analysis of interaction-intensive industrial software systems.

Structure. In Sec. 2, we provide a brief overview of the ASK system. In Sec. 3, we present the hierarchical modeling approach, the realization of important concepts in Reo and, as an example, a more detailed description of the model. The verification issues for an important part of the ASK system at different levels of detail are discussed in Section 4 along with the application of Vereofy to the ASK model.

2 The ASK System

ASK [5] is an industrial software system for connecting people to each other. The system uses intelligent matching functionality in order to find effective connections between requesters and responders in a community. ASK has been developed by Almende [1], a Dutch research company focusing on the application of self-organisation techniques in human organisations and agent-oriented software systems. The system is marketed by ASK Community Systems [5]. ASK provides mechanisms for matching users requiring information or services with potential suppliers. Based on information about earlier established contacts and feedback of users, the system learns to bring people into contact with each other in the most effective way. Typical applications for ASK are workforce planning, customer service, knowledge sharing, social care and emergency response. Customers of ASK include TNT Post, Rabobank and Pfizer. The number of people using a single ASK system varies from several hundreds to several thousands.

System Architecture. The ASK system can be technically divided into three parts: the *web front-end*, the *database* and the *ASK core*. The web front-end acts as a configuration dashboard, via which typical domain data like users, groups, phone numbers, mail addresses, interactive voice response menus, services and scheduled jobs can be created, edited and deleted. This data is stored in a database, one for each deployment instance of the ASK system. The feedback of users and the knowledge derived from earlier established contacts are also stored in this database. Central to our case study, however, is the *ASK core* (see Fig. 1), the "communication engine" of the system, which consists of a quintuple of components: *Reception*, *Matcher*, *Executer*, *Resource Manager* and *Scheduler*. These components handle all communication with and between users of ASK, provide matching functionality and schedule outbound communication and other kinds of jobs.

The "heartbeat" of the ASK core is the *Request loop*, indicated with thick arrows. Requests flow through the request loop until they are fully completed and

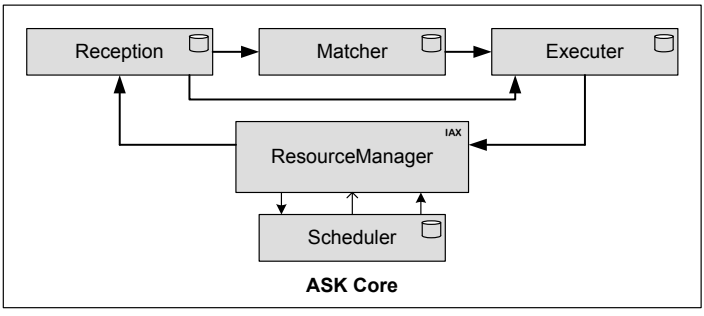


Fig. 1. ASK Core Overview

removed. Requests are primarily initiated by the *Resource Manager* and normally removed by the *Executer*. The request loop itself is *medium and resource independent*. Within the *Resource Manager* component, the loop is separated from the level of media-specific resources needed for fulfilling the request. Telephony connections are handled inside the *Resource Manager* with the help of *Asterisk* (IAX), a telephony development tool-kit. In particular, the individual components in the ASK core perform the following tasks: The *Reception* coordinates the communication process, i.e., the steps to be taken by ASK in order to fulfill a request, while the *Matcher* seeks appropriate context-based participants for a request. The *Executer* determines the best way in which the matching participants can be connected at the time of the request and the *Resource Manager* effectuates and maintains the actual connection between participants. Finally, the *Scheduler* schedules jobs, in particular requests for (outbound) communication between the ASK system and its users.

Component Architecture. The components within the ASK core all have a similar inner structure. Each of them is equipped with an enhanced thread-pool, which is called an *abbey*. The threads within the pool are called *monks*: workers capable of handling specific types of *tasks*, which they get from a single *task queue*. Several types of abbeys are in use. In this paper, we restrict ourselves to a single type of abbey for all components, consisting of a fixed amount of monks and a fixed size task queue. The operation of putting a task in the queue blocks if the queue is full. Tasks in ASK can be either *finite*, in which case a monk executes the task and then is able to perform another task, or *infinite*, in which case the execution of the task never ends (and the monk executing the task is dedicated to it forever). The *hostess* task is a particular example of an infinite task which is present in all components involved in the request loop. Within the hostess task, requests sent to a component are received, converted into tasks, and put into the local task queue in order to be eventually executed by a monk.

Within the *Scheduler* component, which is not involved in the request loop, tasks are put in the task queue based on *job descriptions* in the database (a special task is responsible for this). Here, a job stands for a task with a timestamp

attached to it. The timestamp indicates the time at which the task must be put in the task queue (note that this is different from either starting time or deadline for execution of the task).

Resource Management. An important resource limitation in ASK is the number of *telephony lines* that can be connected at the same time – due to hardware constraints. This limitation is especially relevant in case the *Scheduler* is used to initiate outbound calls to many people at the same time. A special asynchronous communication is used between the *Scheduler* and the *Resource Manager*, which is used to inform the *Scheduler* about the load put on the *Resource Manager*. This load is expressed in terms of a *happiness factor*, ranging from 0 (a heavy load, the *Resource Manager* is "sad") to 1 (no calls at all, the *Resource Manager* is "happy"). To simplify the presentation in this paper, we use a binary happiness factor providing the alternatives "happy" and "sad" only.

Verification Issues. Modeling the ASK system gives rise to a variety of verification issues of interest. On a basic level, this includes the correct functioning of the coordination employed to model the real-world concepts in the Reo network. On a higher level, the behavior of the components acting in concert can be analyzed to ensure that certain assumptions are always adhered to, e.g., the absence of dead-locks or that a particular behavior may only occur in special circumstances. The analysis of liveness properties ensures that certain desirable behavior actually occurs, e.g., that a task will eventually be handled, etc. Of particular interest for the potential refactoring of parts of the system is the question of equivalence of model variants.

3 Top-Down Modeling and Specification of ASK

The ASK core has been written in the ANSI C programming language, while its runtime structure consists of five multi-threaded UNIX processes potentially running on different machines. An appropriate method was needed to translate the "implementation concepts" of the system (processes, threads, shared data structures, functions, function calls, etc.) into the "modeling concepts" of Reo (channels and nodes) and constraint automata (states, transitions, and constraints). We will first provide a brief overview of these modeling concepts.

3.1 Basic Principles of Reo and Constraint Automata

A Reo network consists of a circuit of channels and nodes, providing the coordination for a number of components. The most basic channel is the *synchronous* channel (), passing data from one end to the other. A variant of this channel is the *synchronous drain* channel (), which synchronizes activity at both ends, i.e., one of the ends can only be active if the other is active as well. *Filter* channels () only pass data if their filter condition is satisfied. Asynchronous communication is facilitated by *FIFO* buffer channels () of

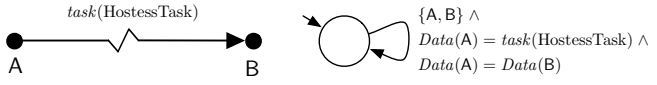


Fig. 2. Customized filter channel and the corresponding constraint automaton

various capacities. The channel ends are either connected to the interface ports of the subcomponents or to Reo nodes, which enforce certain coordination semantics for the connected channel ends. Communication via the default Reo node (●) requires that exactly one of the incoming channel ends is active, i.e., choosing between the available inputs, and that all of the outgoing channel ends are active, i.e., simultaneously copying the received data to all outputs. Route nodes (⊗) behave the same on the input side but will route the input to exactly one of the connected outgoing channel ends. By designating certain nodes as interface ports, the Reo network can be regarded as a component and used as a subcomponent in some higher level network. For further details, we refer to [4].

Constraint automata are used as an operational semantic model for the Reo network, by providing constraint automata for the parts of the network and employing appropriate compositional operations. Customized channels can thus be easily created by providing their behavior as an automaton. As an example, Fig. 2 shows a synchronous Reo filter channel and the corresponding constraint automaton. The transition constraint requires that both ports A and B can only be active simultaneously, that the data transferred at both of them is identical and that it matches the filter condition. The channel will thus only pass messages with the specified content and block otherwise. The behavioral interfaces of components can similarly be specified as constraint automata.

In essence, constraint automata can be regarded as transition systems with a special form of annotations specifying the enabledness of communication actions. The concepts of well-known temporal logics like LTL and CTL can thus be adapted for constraint automata, augmented with special operators allowing the formalization of the data flow via regular expressions over the I/O-activity at the ports and nodes. In this paper, we present $\mathtt{LTL}_{\mathrm{IO}}$ formulas (denoted by Φ) using the standard propositional logic operators as well as the standard operators $\Box$ (Globally), $\Diamond$ (Finally), `X` (next step) and `W` (weak until), referring to state properties as well as the communication activity. For the formulas in the branching-time logic `BTSL` (denoted by Ψ), we mainly use the $\exists\langle\!\langle\alpha\rangle\!\rangle\mathtt{true}$ operator, denoting the existence of an execution prefix such that the communication activity matches the regular expression α. For further details, we refer to [7,9].

As we pointed out earlier, the concepts used in the implementation must be mapped to appropriate concepts in the Reo and constraint automaton framework. For this purpose, the data domain for the messages in the model is structured in such a way as to capture the essential information present in the implementation, e.g. function parameters or return values. Threads, tasks and shared variables are modeled as components with the appropriate connector glue as detailed below, while the various queues in the ASK system are mapped to FIFO

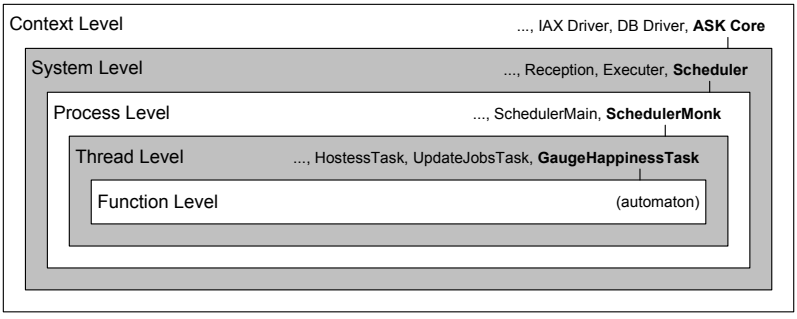


Fig. 3. Abstraction levels in the ASK model and components contained in the level

channels. More complex data structures, such as a hash table for mapping addresses are modeled as well by specifying their behavior as constraint automata.

3.2 Hierarchical Modeling

To adequately capture the implementation concepts of the system in the model, we developed a *hierarchical* modeling method suitable for the modeling of ASK and other systems which rely on similar implementation concepts. We have identified at least five appropriate hierarchical levels of abstraction for use within the Reo network, as shown in Fig. 3.

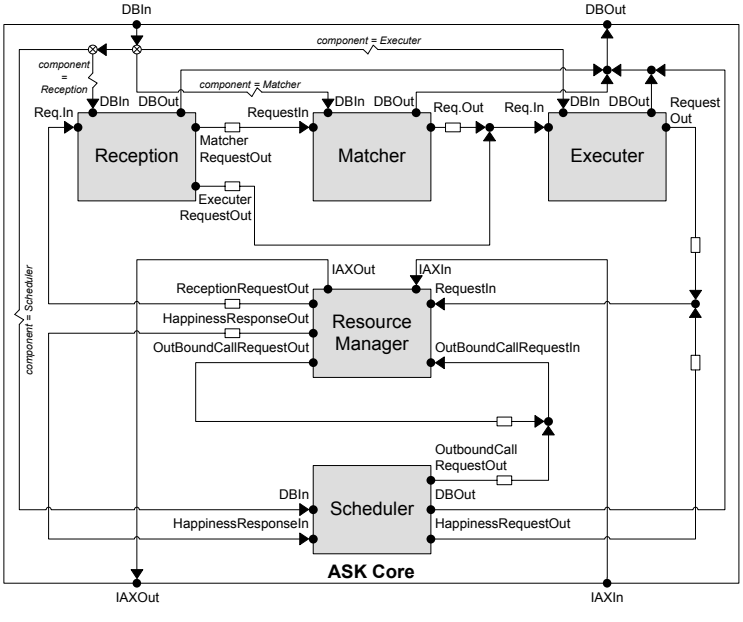


Fig. 4. Reo network of the ASK Core (system level)

At the highest level of abstraction, the *Context Level*, we model the system as an entity interacting with its context – in our case, the ASK database and the telephony IAX hub, which have been modeled as *drivers* for the behavior of the system. At a lower level, the *System Level*, we model the system in isolation, as a network consisting of several processes – the five components of the ASK core. An even lower level, the *Process Level*, focuses on the organization within a process: the Reo network connects threads and shared data structures (e.g., the task queue). At the *Thread Level*, we zoom in onto a single thread within a process – at this level control flow is purely sequential: the Reo network implements low-level control statements like function calls, if-then and while-do, thereby connecting C functions to each other. The lowest *Function Level* can be used (even recursively) to zoom in onto lower levels within the function call tree. In our model of the ASK core, at this level we only modeled the interface behavior via constraint automata.

System level: ASK Core. We start the illustration of our approach with a Reo network at the system level (see Fig. 4). At this level, the Reo network connects the five components of the *ASK Core* shown earlier in an informal manner in Fig. 1. The channels between the components represent UDP communication channels in the real system. The network itself is wrapped into an *ASK Core* component, with interface ports to the context-level components *DB Driver* (top) and *IAX Driver* (bottom).

Process level: Scheduler. As a second step in our illustration, we zoom in onto one of the system level components and consider it at the process level. The purpose of the *Scheduler* process (Fig. 5) in the ASK system is to schedule activities at certain times. In our model, we limit ourselves to the scheduling of outbound calls. In addition, we abstract from the precise timing, i.e., the outbound calls are scheduled in a non-deterministic fashion.

Interface. The interface of the *Scheduler* consists of a port OutboundCallRequest Out for initiating a call, two ports DBIn and DBOut to communicate with the database, as well as two ports for getting an update of the current happiness value of the ASK system. As we explained earlier, the happiness value (either "happy" or "sad") is determined by the *ResourceManager* and reflects the level of congestion of the resources necessary to place a call, e.g., the number of free telephone lines, etc. The *Scheduler* requests an update of the happiness value at a regular time interval via port HappinessRequestOut, with the response arriving at port HappinessResponseIn. It should never initiate an outbound call if the happiness value is "sad".

Structure. The *Scheduler* is composed of several thread components: A single control thread (*SchedulerMain*) and a pool of several worker threads (*SchedulerMonks*). Each *SchedulerMonk* consumes requests to carry out some task from the task queue. While carrying out the task it may put request for the scheduling of a job in the job queue and read or write the shared variable containing the last

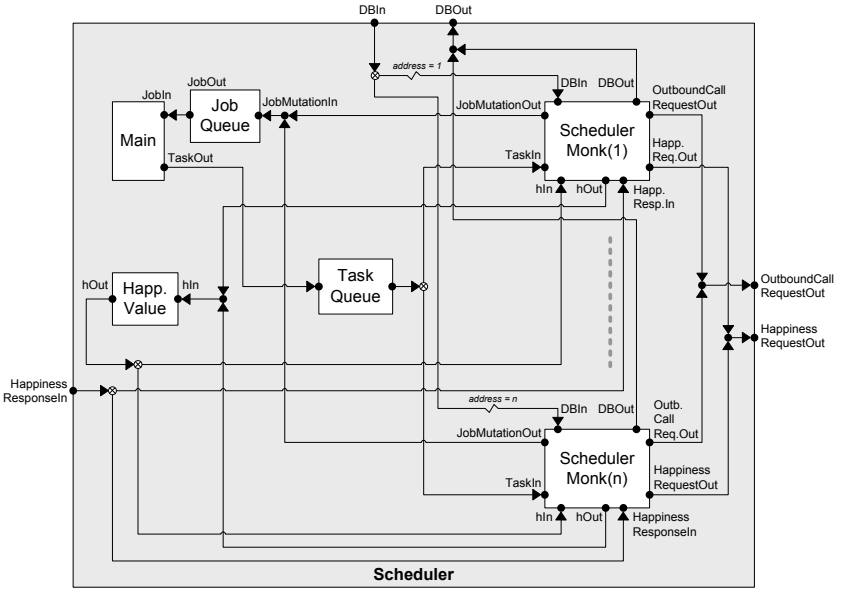


Fig. 5. Reo network for the *Scheduler* (process level)

known happiness value. Furthermore, it may communicate with the rest of the ASK system via the interface ports of the *Scheduler*. The *SchedulerMain* thread is responsible for initially filling the task queue and then transforming the requests from the job queue into appropriate tasks. The Reo network is designed to ensure that external responses reach the appropriate recipient and that attempts of concurrent access are resolved appropriately. The number of *SchedulerMonks* in the worker thread pool and the size of the queues are parametrized in the model. While the *SchedulerMain* thread and the shared variable are modeled directly by providing an appropriate automaton for their behavior, the *SchedulerMonks* are again composed from subcomponents and a connector network.

Thread level: SchedulerMonk. We will now consider the *SchedulerMonk* threads (Fig. 6) – the worker threads of the *Scheduler* – in more detail. They remain idle until a request to carry out some task arrives via port TaskIn. The request is routed via filter channels depending on the value of the requested task to the appropriate task handler. The *HostessTask* continuously monitors the HappinessResponseIn port and updates the stored happiness value in the *Scheduler* via port hOut. The *GaugeHappinessTask* is responsible for requesting an update of the happiness value, the *UpdateJobsTask* is responsible for requesting that jobs are periodically rescheduled and the *JobWrapperTask* is responsible for initiating an outbound call. Apart from the *HostessTask*, which runs continuously and keeps the monk occupied, the other tasks perform their function, may send new jobs to the job queue of the *Scheduler* to ensure future execution,

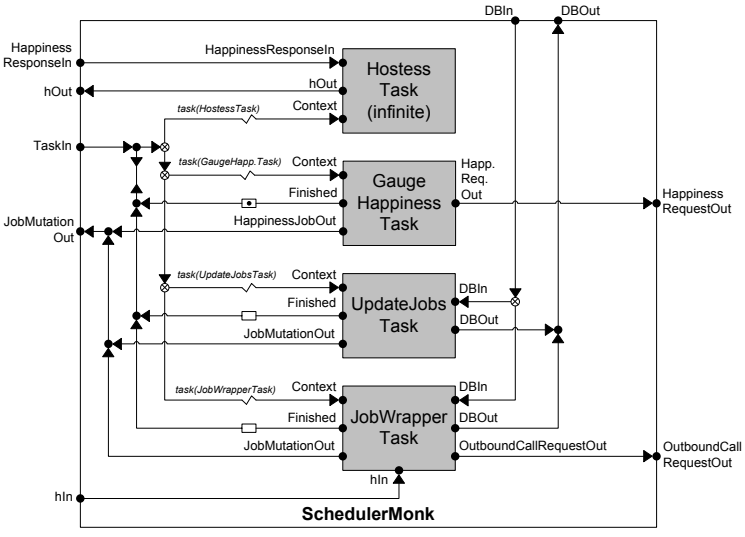


Fig. 6. Reo network of a SchedulerMonk (thread level)

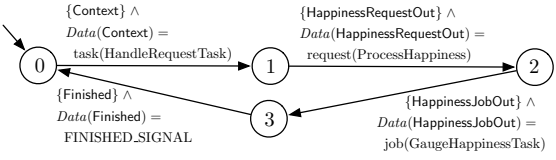


Fig. 7. Automaton for the *GaugeHappinessTask* (function level)

and then finish, freeing the monk to accept a new task. As an example for the task components, which are modeled by constraint automata, consider Fig. 7.

4 Verification of the ASK Model

In the sequel, we detail the specification of the verification issues raised previously in Section 2, starting at the basic level of the coordination mechanisms.

The thread property. When one of the tasks is finished, it sends a token via its Finished port to the attached FIFO buffer. These buffers are connected via a synchronous drain channel (⊶⊷) to the TaskIn port and the router/filter network responsible for delivering the task request to the appropriate task component. This ensures that a new task can only be accepted if there is a token in one of the buffers, and that the token is consumed when the task is accepted. The purpose of this coordination pattern is to ensure that inside a *SchedulerMonk* at most one of the tasks can be active at the same time. Otherwise, a single thread could exhibit concurrent execution, contradicting the

fact that in the ASK system threads are the basic units of concurrent execution and thus the model would allow significant spurious behavior. By using the observation that a task is active if the control location of the corresponding automaton is not in the initial, idle state (e.g., see Fig. 7) where the task waits for activation, we can specify the *thread property* as the $\texttt{LTL}_{\text{IO}}$ formula

$$\Phi_{\text{TP}} = \bigwedge_{t \in Tasks} \square\Big(\text{“Task } t \text{ is not idle”} \quad \longrightarrow \bigwedge_{t' \in Tasks \setminus \{t\}} \text{“Task } t' \text{ is idle”}\Big)$$

where *Tasks* denotes the set of the tasks. Formula Φ_{TP} requires that, for all tasks, whenever a task is not idle, i.e., active, all other tasks are idle, i.e., inactive.

To further analyze the coordination pattern used to ensure the thread property and correct routing of tasks in more detail, the following formulas specify that receipt of a new task message via port TaskIn may only happen if there is a token in one of the buffers (Φ_{M1}, $\texttt{LTL}_{\text{IO}}$), that after a new task message has been received the token was consumed and all buffers are empty (Φ_{M2}, $\texttt{LTL}_{\text{IO}}$), that the correct task is activated (Φ_{M3}, $\texttt{LTL}_{\text{IO}}$) and that a task may only send the finish signal when it has been previously activated (Ψ_{M4}, BTSL):

$$\Phi_{\text{M1}} = \square\ (\text{“TaskIn active”} \longrightarrow \neg \text{“all buffers empty”})$$

$$\Phi_{\text{M2}} = \square\ (\text{“TaskIn active”} \longrightarrow \texttt{X}\ \text{“all buffers empty”})$$

$$\Phi_{\text{M3}} = \bigwedge_{t \in Tasks} \square\big(\text{“}\mathit{Data}(\text{TaskIn}) = \mathit{task}(t)\text{”} \longrightarrow \text{“Context of } t \text{ is active”}\big)$$

$$\Psi_{\text{M4}} = \bigwedge_{t \in Tasks} \neg\exists\langle\!\langle\ \text{“Context of } t \text{ inactive”}^*; \text{“Finished of } t \text{ active”}\rangle\!\rangle\ \texttt{true}$$

Simplification of task coordination. As these properties can be shown to hold for the *SchedulerMonk*, we can conclude that at most one of the three FIFO channels storing the “finished” token is full at the same time (Φ_{B}, $\texttt{LTL}_{\text{IO}}$):

$$\Phi_{\text{B}} = \bigwedge_{b \in Buffers} \square\Big(\text{“}b \text{ is full”} \quad \longrightarrow \bigwedge_{b' \in Buffers \setminus \{b\}} \text{“}b' \text{ is empty”}\Big)$$

We can thus simplify the coordination pattern by replacing the three FIFO channels by a single FIFO channel. We used the bisimulation checker of Vereofy to show that the original and the simplified variant of the *SchedulerMonk* are bisimulation equivalent for the behavior at the interface ports. This shows that the refactoring of the coordination circuit was valid, giving the strong guarantee that the same branching and linear-time properties – that don't refer to internals of the *SchedulerMonk* – hold for both variants. In the ASK model, we can thus replace the original model of the *SchedulerMonk* by the refactored variant.

Happiness. As a further example, we now analyze the handling of the happiness value. The *Scheduler* is supposed to refrain from requesting some outbound call if the happiness value is “sad”. As the happiness value is determined by the *ResourceManager*, the *Scheduler* regularly requests an update of the happiness value. It is clear that there can be a race condition where the *Scheduler* bases

its decision to initiate an outbound call on its most current information of the value, while concurrently the value is changed in the *ResourceManager*. We thus consider here only the aspect of the handling of the happiness value based on the knowledge of the *Scheduler*. As the decision to initiate an outbound call is made in one of the *SchedulerMonk* worker threads, we will first consider the handling of the happiness value at the *SchedulerMonk* level. The desired behavior can be specified by the following properties: Before the *SchedulerMonk* sends an OutboundCallRequestOut, it has previously read a happiness value via port hIn and the received value was "happy" (Ψ_{H1}, BTSL) and that between two OutboundCallRequestOut, the monk receives a fresh happiness value via port hIn where the value was "happy" (Φ_{H2}, LTL_{IO}).

$$\Psi_{\text{H1}} = \neg\exists\langle\!\langle \text{“hIn inactive} \vee \mathit{Data}(\text{hIn}) \neq \texttt{happy”}^*; \text{“OutboundCallRequestOut”}\rangle\!\rangle \texttt{true}$$

$$\Phi_{\text{H2}} = \square\big(\text{“OutboundCallRequestOut”} \longrightarrow \texttt{X}(\neg\text{“OutboundCallRequestOut”}\ \texttt{W}\ \text{“hIn and } \mathit{Data}(\text{hIn}) = \texttt{happy”})\big)$$

Checking these formulas using Vereofy show that the *SchedulerMonk* behaves as expected. One level above, at the *Scheduler* process level, we would like a similar property to hold as well, i.e., that an OutboundCallRequestOut may only happen if the last HappinessResponseIn contained a "happy" value, as formalized by the BTSL formula Ψ_{H3}, where step signifies an arbitrary step in the system:

$$\Psi_{\text{H3}} = \neg\exists\langle\!\langle \texttt{step}^*; \text{“HappinessResponseIn and } \mathit{Data}(\text{HappinessResponseIn} = \texttt{sad})\text{”}; \text{“}\neg\ \text{HappinessResponseIn”}^*; \text{“OutboundCallRequestOut”}\rangle\!\rangle \texttt{true}$$

Trying to verify this formula with the model checker shows that this property is violated for the *Scheduler*. An analysis of the produced counterexample trace shows that it is possible that a *SchedulerMonk* determines that the currently known happiness value is "happy" and commences with preparations to initiate the outbound call, e.g., querying the database for further information, etc., before actually sending the OutboundCallRequestOut. During this time it is then possible for a fresh HappinessResponseIn to arrive, now with the happiness value being "sad" and the *SchedulerMonk* will go ahead with sending the OutboundCallRequestOut despite this new information. An examination of the implementation of the ASK system showed that such a race condition was indeed present in the version of the system considered in this case study. This issue could be dealt with by adding checks to ensure that the JobWrapperTask has up-to-date information before requesting the call.

Liveness. Another aspect of interest are questions of liveness in the system model at the different levels, i.e., that the system remains responsive. At the level of the *SchedulerMonk* worker threads, the primary interest consists of verifying that the processing of the assigned tasks is able to finish and the *SchedulerMonk* is available again to process some new task – except in the case of being assigned the *HostessTask*, which is by construction infinite and thus continuously occupies the thread, formalized by the LTL_{IO} formula Φ_{L1}:

$$\Phi_{\text{L1}} = \Box\big(\text{“}\mathsf{TaskIn}\text{ and }\mathit{Data}(\mathsf{TaskIn}) \neq \mathit{task}(\text{HostessTask})\text{”} \longrightarrow \mathtt{X}\,\Diamond\,\text{“}\mathsf{TaskIn}\text{ is enabled”}\big)$$

When checking property Φ_{L1} for the *SchedulerMonk* component in isolation, i.e., with an unspecified environment, the model checker uses the mild assumption that communication at the interface ports will not be continuously blocked by the environment. The atomic proposition “TaskIn is enabled” then labels those states where the reception of a new task is not blocked by the monk. Having verified the property above, we can thus conclude that the *SchedulerMonk* will indeed finish processing its task, as long as the environment provides appropriate interaction at the interface ports, i.e., that the monk will not deadlock by itself. On the *Scheduler* level, we need to use fairness assumptions to rule out pathological cases, e.g., where only one of the monks is active all the time although the other monks would be able to be active. Thus we require strong fairness for all the relevant components of the *Scheduler*, i.e., for the *Main* thread and each of the *Monks* it is required that if there is infinitely often the possibility of making a step, then it infinitely often makes a step. In addition, we require a similar fairness for the internal steps of the thread and job queues which are modeled by the concatenation of multiple simple FIFO buffers, i.e., that a data value will eventually move from the input end of the queue to the output end if possible. Using Vereofy, we proved the realizability of the fairness assumption, i.e., that there are indeed executions in the system that are fair. The analogous property to formula Φ_{L1} at the *SchedulerMonk* level is then LTL_{IO} formula Φ_{L2} at the *Scheduler* level:

$$\Phi_{\text{L2}} = \bigwedge_{m \in Monks} \Box\big(\text{“}\mathsf{TaskIn}\text{ of monk }m\text{ and }\mathit{Data}(\mathsf{TaskIn}) \neq \mathit{task}(\text{HostessTask})\text{”} \longrightarrow \mathtt{X}\,\Diamond\,\text{“Monk }m\text{ is finished”}\big)$$

We continue analyzing the functionality of the *Scheduler* process. The *Main* thread is expected to continuously process requests for jobs placed in the job queue by the monks during the execution of some previous task, and transform them into tasks and place them into the task queue. This can be formalized by the LTL_{IO} formula Φ_{L3}, implying that there is a steady stream of tasks being generated and – due to the finite size of the task queue – actually processed:

$$\Phi_{\text{L3}} = \Box\Diamond\big(\text{“}\mathsf{TaskOut}\text{ of }\mathit{Main}\text{”}\big)$$

Furthermore, there should be a steady stream of outbound call requests if the *Scheduler* gets a “happy” response and from then on there are no further “sad” responses, formalized by the LTL_{IO} formula Φ_{L4}:

$$\Phi_{\text{L4}} = \Diamond\big(\text{“}\mathsf{HappinessResponseIn}\text{”} \wedge \Box(\text{“}\mathsf{HappinessResponseIn}\text{”} \longrightarrow \text{“}\mathit{Data}(\mathsf{HappinessResponseIn}) = \mathtt{happy}\text{”})\big) \longrightarrow \Box\Diamond\,\text{“}\mathsf{OutboundCallRequestOut}\text{”}$$

4.1 Verification of the ASK Model Using Vereofy

In Vereofy the automata at the lowest level of the ASK model are specified in CARML, while the components at the higher levels, consisting of subcomponents

and a Reo network of channels connecting them, are generated by means of RSL scripts[1]. As a guarded command language, CARML allows the concise specification of the component behavior amenable to an efficient symbolic representation. RSL scripts mainly provide a versatile way of instantiating subcomponents and channels and join them together, as well as elegant mechanisms for generating parametrized model variants which in some case are abstractions from the full model. Overall, the ASK main model in its simplest version, where all parameters have been set to small reasonable values, consists of 334 component instances and channels (with 66 different types) from which 79 constitute the *Scheduler*, 38 the *Matcher*, 40 the *Reception*, and 141 the *ResourceManager*. The remaining channels form the coordinating network.

The ASK model is – to our best knowledge – the largest and most complex Reo network to date and thus provided a challenging test case for the Vereofy model checker. Internally, Vereofy relies on a symbolic representation based on binary decision diagrams (BDDs) of the automata for the components and the network. The symbolic representation of the ASK model at full detail uses 393 boolean variables to encode the state space of the composite system. There are 1018 distinct communication points in the model, i.e., interface ports or Reo nodes, with the transferred data values at each point being encoded by 11 boolean variables (for 290 distinct message values). Vereofy employs well known dynamic reordering techniques, as well as special heuristics for the Reo framework to identify redundancies and gain a compact representation. The number of reachable states of the processes which we could build in full detail reaches a magnitude of up to 10^{10} states.

In the sequel, we present some statistics related to the modeling and verification as presented in Sec. 3. The table below shows the size (number of BDD nodes) of the internal representation for the components detailed in the previous section, as well as the time spent to pre-compute a favorable variable ordering for the boolean variables, which can then be reused to build the internal representation in the time shown in the third column. Here, the *Scheduler* has been built with two monks and in three variants where the capacity of the task queue is either one, two, or three (tq1, tq2, and tq3). All computations were performed on an AMD Athlon 64 X2 Dual Core Processor 5000+ (2.6 GHz) with 8GB of RAM, running Ubuntu Linux.

Component	**BDD nodes**	**Build time**	**Reorder time**
GaugeHappinessTask	59	< 1s	-
SchedulerMonk	15319	< 1s	-
SchedulerMonk (simplified)	8632	< 1s	-
Scheduler (tq1)	71327	2, 7s	264s
Scheduler (tq2)	797846	7, 7s	647s
Scheduler (tq3)	3147036	10, 0s	2292s

[1] The CARML/RSL code for the ASK model can be found at `http://wwwtcs.inf.tu-dresden.de/ALGI/ASK-FMICS2011.tgz`

For the verification of the ASK system model we made use of Vereofy's model checking engines for $\texttt{LTL}_{\text{IO}}$ and BTSL as well as the bisimulation checker [7]. All properties presented in Sec. 3 could be shown to hold, except those where it was already explained why they do not hold. The properties for the *SchedulerMonk* could all be checked in less than one second, with the bisimulation check of the equivalence with the simplified variant taking 16 seconds. The table below summarizes the verification times for the considered *Scheduler* variants.

Property	Verification (tq1)	Verification (tq3)	Verification (tq3)
Ψ_{H3}	0, 5s	7, 3s	7, 8s
realizability of fairness	5, 7s	324s	2713s
Φ_{L2}	5, 3s	329s	2913s
Φ_{L3}	2, 4s	200s	3034s
Φ_{L4}	11, 0s	816s	3533s

We were able to identify and fix several implementation inefficiencies in Vereofy and develop further optimization heuristics to be able to handle models of this scope. As the full composition of all the layers of the model at the full level of detail remained elusive, we have employed abstraction techniques such as automatic abstraction from concrete data values, the replacement of components by ones with more generic behavior, to nevertheless be able to gain insights at these levels that do not rely on the detailed behavior. Further improvements of the techniques to cope with the state space explosion as well as support for additional convenient automatic abstraction techniques remain as future work.

5 Conclusion

Our aim was to show that the exogenous modeling approach using coordination languages facilitates the hierarchical decomposition of complex systems into manageable components at different layers of abstraction. In particular, the communication and coordination aspects arising in distributed and complex systems like ASK can be modeled in an explicit and intuitive manner. This separates coordination from computation and enables the independent top-down refinement of both the components and the coordination glue code. In this regard, the modeling phase itself proved useful to gain insight into the parts of the ASK system and their complex interactions. The modeling of the ASK system was supported by the input formalisms of Vereofy allowing the comprehensive specification of complex component behavior and the scripted composition of the Reo network for comfortable parameterization and generation of model variants. Tool support for verification provided by Vereofy then was crucial to gain further insights, both in the form of exploration of the system behavior (random traces, visualization of the communication, exploration of the model using counter-examples/witnesses for temporal logic formulas) as well as for the comfortable specification and formal verification of interesting properties, as exemplified in Section 4. Instead of only allowing the verification of components by themselves and the coordination patterns in isolation, Vereofy offered an integrated verification approach supported by the underlying common constraint automata framework.

As a result of the insights gained in this case study, Almende has taken three initiatives. Firstly, bugs in the ASK system have been eliminated based on the outcomes of the verification. Secondly, the modeling approach presented in this paper has been adopted for the hierarchical modeling of other critical parts of the ASK system, primarily in order to gain further insight in potential refactorings of the code leading to an improved modular structure. However, we certainly expect to find more problems and other issues through the application of Vereofy. Finally, ideas have been sketched at Almende to apply high-level Reo circuits as coordination glue for lower-level modules, as soon as a better modular structure has been achieved.

References

1. The Almende research organization, `http://www.almende.com/`
2. Alur, R., de Alfaro, L., Grosu, R., Henzinger, T.A., Kang, M., Kirsch, C.M., Majumdar, R., Mang, F.Y.C., Wang, B.-Y.: Jmocha: A model checking tool that exploits design structure. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE), pp. 835–836. IEEE Computer Society Press, Los Alamitos (2001)
3. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design 15(1), 7–48 (1999)
4. Arbab, F.: Reo: A Channel-Based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
5. ASK community systems, `http://www.ask-cs.com/`
6. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A Uniform Framework for Modeling and Verifying Components and Connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 247–267. Springer, Heidelberg (2009)
7. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal Verification for Components and Connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009)
8. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling Component Connectors in Reo by Constraint Automata. Science of Computer Programming 61, 75–113 (2006)
9. Blechmann, T., Klein, J., Klüppelholz, S.: Vereofy User Manual. Technische Universität Dresden (2008–2011), `http://www.vereofy.de/`
10. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer 2(4), 410–425 (2000)
11. Gößler, G., Sifakis, J.: Component-based construction of deadlock-free systems. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 420–433. Springer, Heidelberg (2003)
12. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23, 279–295 (1997)
13. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Technical report, CWI, Amsterdam, The Netherlands (1998)

Modeling and Verifying Timed Compensable Workflows and an Application to Health Care

Ahmed Shah Mashiyat, Fazle Rabbi, and Wendy MacCaull

Centre for Logic and Information
St. Francis Xavier University
Antigonish, Canada
{x2008ooc,x2010mcf,wmaccaul}@stfx.ca

Abstract. Over the years, researchers have investigated how to provide better support for hospital administration, therapy and laboratory workflows. Among these efforts, as with any other safety critical system, reliability of the workflows is a key issue. In this paper, we provide a method to enhance the reliability of real world workflows by incorporating timed compensable tasks into the workflows, and by using formal verification methods (e.g., model checking). We extend our previous work [1] with the notion of time by providing the formal semantics of *Timed Compensable WorkFlow nets* (CWF_T-nets). We extend the graphical modeling language of *Nova WorkFlow* (a workflow management system currently under development) to model CWF_T-nets and enhance *Nova WorkFlow*'s automatic translator to translate a CWF_T-net into DVE, the modeling language of the distributed LTL model checker DiVinE. These enhancements provide a method for rapid (re)design and verification of timed compensable workflows. We present a real world case study for Seniors' Care, developed through collaboration with the local health authority.

Keywords: Workflow System, Compensable Task, Time Constraint, Distributed Model Checking, Health Services Delivery.

1 Introduction

Medical and health services delivery processes are complex and have many stages that require collaboration and coordination among several professionals and departments, which may leave the processes insufficiently defined. Medical errors including miscommunication between clinical and other health care professionals, incorrect drug and other therapeutic administration, and miscalculated drug doses are costly for the system, may adversely affect the patient, and may lead to patient's death. Workflow Management Systems (WfMS) can contribute to the development of efficient and clearly defined health care processes. Reliability of such safety critical processes is of vital inportance.

Model checking is an automatic formal verification method, which has great potential for verifying models of complex and distributed business processes. For the most part, tools for model checking can verify only the correctness of

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 244–259, 2011.

statements dealing with relative time, such as "*Eventually the patient receives a certain treatment*". However, to save the patient's life, it should be verified that the medical process satisfies "*The patient receives a certain treatment within half an hour*" [2]. For such safety-critical applications, quantified time notions including time instance and duration must be taken into account while model checking.

Usually, a major part of a workflow definition defines "normal" behaviors in response to anticipated events and/or results. An exception in the workflow is an "abnormal" behavior. Exceptions in health services delivery workflows may cover a wide variety of events, not limited to medical emergencies, depending on the application context and workflow design decisions [3]. An effective recovery mechanism for these exceptions in a workflow management system is crucial for its success. Time constraints on the recovery mechanism of a workflow make the model checking problem even more difficult. A *compensable transaction* is a type of transaction whose effect can be semantically undone even after it has been committed [4]. As an extended model of ACID (Atomic, Consistent, Isolated, Durable) transactions, *compensable transactions* help support the "abnormal" behaviors of workflows.

In this paper, we extend the semantics of Compensable Workflow nets, and the components of the graphical Compensable Workflow Modeling Language [1] with the notion of time. We also extend the automatic translators [1], [2] and integrate it into Nova WorkFlow[1] [5], a Workflow Management System currently under development. Nova WorkFlow allows the user to graphically design a timed compensable workflow, specify an LTL property for the model, and automatically generate the DVE code of the model for verification by DiVinE. We show the usefulness of our method with a real world case study.

The remainder of this paper is as follows: Section 2 presents some background topics; Section 3 describes Timed Compensable Workflow nets, its underlying formalism, and its graphical modeling components; Section 4 presents a verification method for Timed Compensable Workflow nets; Section 5 describes a case study, and Section 6 concludes the paper, discusses related work, and offers some directions for future work.

2 Preliminaries

This section provides background information about workflows and their time constraints, and compensable transactions.

2.1 Workflow and Time Constraints

For control purposes, a workflow may be viewed as an abstraction of the real work under a chosen aspect that serves as a virtual representation of the actual work. Therefore, a workflow is a collection of activities and the dependencies among those activities. The activities correspond to individual tasks in a business

[1] `http://logic.stfx.ca/software/nova-workflow/Overview/`

process. Dependencies determine the execution sequence of the activities and the data flow among these activities.

When we talk about activities (or tasks) and the dependencies among them, time plays an important role. Several explicit time constraints have been identified for the time management of an activity [6]. *Duration* is the time span required to finish a task. *Forced start time* prohibits the execution of a task before that certain time. *Deadline* is a time based scheduling constraint which requires that a certain activity be completed by a certain time. A constraint, which forces an activity to be executed only on a certain fixed date, is referred to as a *fixed date constraint*. *Delay* is the time duration between two subsequent activities. Besides these explicit time constraints, some time constraints follow implicitly from the control dependencies and activity durations of a workflow model. They arise from the fact that an activity can start only when its predecessor activities have finished. Such constraints are called *structural time constraints* since they abide by the control structure of the workflow [6]. The concept of *relative constraint* which limits the time distance (duration) between the starting and ending instants of two non-consecutive workflow activities can be found in [7]. In this work, we consider both duration and delay constraints which together are capable of modeling the timing aspects of almost any workflow [8]. Duration and delay are expressed by integer values following the Gregorian calendar, i.e., year, month, week, day, hour, and minute.

2.2 Compensable Transaction

A compensable transaction refers to a transaction with the capability of withdrawing its effect after its commitment, if an error occurs. A compensable transaction may be described by its external state. In [4] we find a finite set of eight independent states, called *transactional states*, which can be used to describe the external state of a transaction at any time. These transactional states (and their abbreviation) are idle, active, aborted, failed, successful, undoing, compensated, and half-compensated, where successful, aborted, failed, compensated, and half-compensated are terminal states. Before activation, a compensable transaction is in the idle state. Once activated, the transaction eventually moves to one of the terminal states. A successful transaction has the option of moving to the undoing state. If the transaction can successfully undo all its partial effects it goes to the compensated state, otherwise it goes to the half-compensated state.

The transactional composition language, t-calculus, was proposed to describe reliable systems composed of compensable transactions. In addition, it provides flexibility and specialization, commonly required by business process management systems, with several alternative flows to handle exceptional cases. The syntax of t-calculus incorporates several operators to compose compensable transactions: Sequential Composition ($S \ ; \ T$), Parallel Composition ($S \ || \ T$), Internal Choice ($S \sqcap T$), Speculative Choice ($S \otimes T$), Alternative Forwarding ($S \rightsquigarrow T$), Backward Handling ($S \trianglerighteq T$), Forward Handling ($S \triangleright T$), and Programmable Composition ($S \circledast T$), where S and T represent arbitrary compensable transactions [4].

3 Modeling Timed Compensable Workflow with CWF$_T$-Nets

In this section, we extend the semantics of Compensable Workflow nets (CWF-nets) [1] with the notion of time. We use a Petri net based formalism to provide a sound mathematical foundation, and provide the graphical modeling components of CWF$_T$-nets.

3.1 Timed Compensable Workflow Nets (CWF$_T$-Nets)

We start by defining CWF-nets and its elements based on [1] and [5] and gradually build the semantics of CWF$_T$-nets.

An atomic task is an indivisible unit of work. Atomic tasks can be either compensable or uncompensable. Generally, if activated, an atomic uncompensable task always succeeds [9], while an atomic compensable task either succeeds or fails and is compensated. When a task executes, it performs some actions, and its execution may depend on some conditions. The formal definition of pre-condition and action are given below:

Definition 1. *A term, σ, is defined using BNF as: $\sigma ::= c \mid \chi \mid \sigma \oplus \sigma$, where $\oplus \in \{+, -, \times, \div\}$, c is a natural number and χ is a (natural) variable.*

A pre-condition is a formula, ψ^p, is defined as $\psi^p ::= \sigma \diamond \sigma \mid (\psi^p \uplus \psi^p)$, where $\diamond \in \{<, \leq, >, \geq, ==\}$, $\uplus \in \{\&\&, ||\}$ and σ is a term. An action, ψ^a, is an assignment defined as $\psi^a ::= v = \sigma$; v is called a mapsTo variable and σ is a term.

A compensable task can be composed with other compensable tasks using the t-calculus operators.

Definition 2. *A compensable task, Γ_c, is recursively defined using BNF as: $\Gamma_c ::= \tau_c \mid (\Gamma_{c_1} \odot \Gamma_{c_2})$, where τ_c is an atomic compensable task, which has a set of pre-conditions $\{\psi_i^p\}$ and sets of actions $\{\psi^a\}$ (forward) and $\{\psi^{a\prime}\}$ (compensation) associated to it, and $\odot \in \{;, ||, \sqcap, \otimes, \rightsquigarrow\}$ is a t-calculus operator.*

Note that, in this paper, we assume if activated, an atomic compensable task τ_c either completes successfully or fully compensates. Therefore, the backward handling operator ($\trianglerighteq$), forward handling operator ($\triangleright$) and programmable compensation operator ($\circledast$) from [1] are omitted. Any task can be composed with uncompensable and/or compensable tasks to create a new task. As above, a task may be considered as a formula; subtasks correspond to subformulas.

Definition 3. *A task, Γ, is recursively defined using BNF as: $\Gamma ::= \tau \mid \Gamma_c \mid (\Gamma_1 \ominus \Gamma_2) \mid (\Gamma_1)^+$, where τ is an uncompensable atomic task, which has a set of pre-conditions $\{\psi^p\}$ and a set of actions $\{\psi^a\}$ associated to it; Γ_c is a compensable task; $\ominus \in \{\wedge, \vee, \times, \bullet\}$ is a binary operator. $^+$ is an unary operator for loops (iteration), and Γ^+ denotes that Γ executes at least once if activated and iterates as long as its pre-conditions are true.*

Any task which is built up from the operators $\{\wedge, \vee, \times, \bullet\}$ is deemed as uncompensable. Thus if Γ_1 and Γ_2 are compensable tasks, then $\Gamma_1;\Gamma_2$ denotes another compensable task while $\Gamma_{c_1} \bullet \Gamma_{c_2}$ denotes a task consisting of two distinct compensable subtasks. The control flow operators $\wedge, \vee$, and $\times$ as well as the t-calculus operators $||, \sqcap, \rightsquigarrow$ and $\otimes$ are associative [5].

Definition 4. *A Compensable Workflow net (CWF-net) is a tuple (i, o,* $\mathbb{T}$*,* $\mathbb{T}_c$*, F) such that:*

- *i is the input condition and* o *is the output condition,*
- $\mathbb{T}$ *is a set consisting of atomic tasks, split tasks, and join tasks,*
- $\mathbb{T}_c \subseteq \mathbb{T}$ *is a set consisting of the compensable tasks,*
- $F \subseteq (\{i\} \times \mathbb{T}\ ) \cup (\mathbb{T} \times \mathbb{T}) \cup (\mathbb{T} \times \{o\})$ *is the flow relation (for the net).*

The elements of a CWF-net (i.e., tasks, input condition, output condition and flow relation) are called workflow components. It is convenient to represent a large CWF-net using smaller CWF-nets, each representing a subnet. A placeholder for the subnet is known as a *composite task*. The first compensable subtask of a compensable task is called the initial subtask; the compensation flow from the initial subtask is directed to an uncompensable task or to the output condition which follows the compensable task; every task in a workflow is on a directed path from i to o.

We used a Petri net based formalism to define atomic tasks, split tasks, and join tasks in [1] [5], where no time constraint is considered. There a CWF-net could be represented by a Petri net. In this paper, we use a modified version of time Petri nets (which we call Explicit Time Petri nets [10]). In Explicit Time Petri nets, we abstract the time granularities of time Petri nets to range over integers rather than real numbers, and use a hybrid semantics based on both the *weak time semantics* and the *strong time semantics* of time Petri nets [11].

Definition 5. *An Explicit Time Petri Net (ETPN) is a tuple,* $PM = \langle P, T_P, F, M_0, m\rangle$ *where: P, F,* M_0 *denote a set of places, a set of arcs and an initial marking, respectively (as in ordinary Petri nets);* $T_P = T_s \cup T_w = \{t_1, t_2, ..., t_n\}$ *is a finite non-empty set of transitions, each of which is associated with a set of pre-conditions* $\{\psi^p\}$*, and a set of actions,* $\{\psi^a\}$*,* T_s *is a set of strong transitions, and* T_w *is a set of weak transitions; m is a mapping for time constraints,* m*:* $T_P \rightarrow D_1 \times D_2$*, where* D_1, D_2 *are sets of positive integers representing delays and durations respectively.*

An ETPN has one global clock to simulate the absolute time, *now*; delay and duration of transition t_i are simulated by a clock, local to the transition. The local clocks of transitions are synchronized with the global clock.

A transition t is *enabled* iff $\forall_p \in {}^\bullet t$, M(p)$\geq$1, where ${}^\bullet t$ represents the set of input places of t (similarly, $t^\bullet$ represents the set of output places of t). Some transitions may be enabled by a marking M, but not all of them may be allowed to fire due to the firing constraints of transitions (delays and durations). A transition t (with $d_1 \in D_1$ and $d_2 \in D_2$) enabled by marking M at absolute

time now_{enable} cannot fire before the absolute time ($now_{enable} + d_1$) and can fire any time non-deterministically between ($now_{enable} + d_1$) and ($now_{enable} + d_1 + d_2$), unless disabled by the firing of some other transition. If transition t_i fires, it leads the system to another state, at any time between ($now_{enable} + d_1$) and ($now_{enable} + d_1 + d_2$) (inclusive). An enabled strong transition must fire after ($now_{enable} + d_1 + d_2$) is elapsed, and an enabled weak transition will become disabled after that time. As the global clock is explicitly storing the time (as *now*), we are able to verify quantified properties involving time instances.

Definition 6. *An atomic timed uncompensable task τ is a tuple (E, s) such that:*

- *E is an ETPN, as shown in Fig. 1 (left) where t_1 is a strong transition, which will ensure that the task will finish during its assigned duration (d_2);*
- *s is a set of unit states {idle, active, successful}; the unit state idle indicates that transition t_1 in E is not yet enabled and the delay time (d_1) has not elapsed, the active state indicates that t_1 is enabled and the delay has elapsed, and the successful state indicates that t_1 has fired and produced a token in the place p_suc.*

Note that the unit states of a task are different from the state of an ETPN where a state is determined by the marking (recall a marking is a function from the set of places to the nonnegative integers) of its places and the global clock time. Before we formalize the notion of atomic compensable task we note that compensation duration (d_3) refers to the maximum time required to compensate a failed task plus the duration for the task.

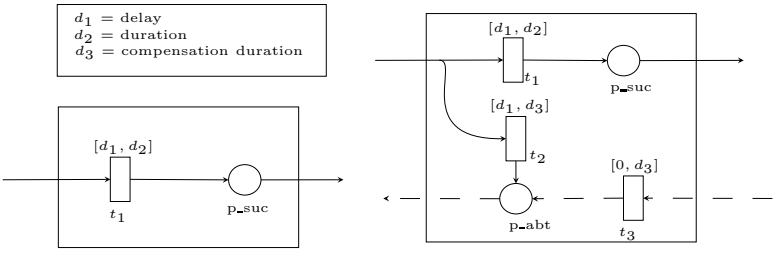


Fig. 1. Uncompensable (left) and compensable (right) atomic tasks in an ETPN

Definition 7. *An atomic timed compensable task τ_c is a tuple (E_c, s_c) such that:*

- *E_c is an ETPN as shown in Fig. 1 (right) where t_1 is a weak transition, and t_2 and t_3 are strong transitions; as a weak transition, t_1 will ensure that the task is compensated if it is not finished within its assigned duration;*
- *s_c is a set of unit states {idle, active, successful, undoing, aborted}, where*
  - *idle indicates that transitions $t_1, t_2,$ and t_3 are disabled, the delay time of t_1, and t_2 have not elapsed and there is no token in p_suc and p_abt;*
  - *active indicates that the transitions t_1, t_2 are enabled but their delay times have not elapsed;*

- *successful indicates that there is a token in place p_suc and the delay time has elapsed;*
- *undoing indicates that the transition t_3 is enabled and the compensation duration time has not elapsed;*
- *aborted indicates that there is a token in place p_abt.*

The task τ_c transits to the unit state *active* after getting a token in the input place of transition t_1. The token has to wait in the input place until d_1 has expired to make the transition *firable* (a transition is *firable*, if it can fire). Transition t_1 has d_2 time to get the work done and produce a token in *p_suc*, representing the unit state *successful*. If it fails to finish the work within d_2, the transition becomes disabled and transition t_2 has to compensate the task failure. Transition t_2 has $d_3 - d_2$ time to compensate the task and produce a token in *p_abt* representing the unit state *aborted*; d_3 is always greater than d_2. Note that, both "firable" transitions can fire at any time within their assigned time durations. The unit state *aborted* indicates an error occurred performing the task or the assigned duration time for the task has elapsed and the effects can be successfully removed. The compensation (backward) flow is started from this point. Petri net representation of *join* tasks, *split* tasks, and compound tasks can be found in [5].

Now we are ready to provide the formal definition of CWF_T-nets.

Definition 8. *A Timed Compensable Workflow net (CWF_T-net) is a tuple (C_N, Tick) such that:*

- *C_N is a CWF-net with atomic timed compensable and uncompensable tasks,*
- *Tick is a global clock that simulates the absolute time now; an enabled Tick may or may not increase the value of now by 1 (depending on whether or not the time has elapsed). Tick is disabled iff there exists a transition t in the CWF_T-net which is firable and its duration time has already elapsed: Formally, Tick is disabled iff, $\exists t(firable(t) = true$ and $(now - (enabled(t) + delay(t))) \geq duration(t))$.*

3.2 Graphical Representation of CWF_T-Net

In [1] we described a graphical workflow modeling language, the Compensable Workflow Modeling Language (CWML), with which one can model a workflow with compensable tasks composed using t-calculus operators. Here we extend this graphical language with time, calling it $CWML_T$. Fig. 2 shows the modeling elements of $CWML_T$, where τ stands for a timed uncompensable task and τ_c stands for a timed compensable task.

A workflow specification in $CWML_T$ is a set of timed compensable workflow nets (CWF_T-nets) which form a hierarchical structure. Tasks are either timed atomic compensable tasks, timed atomic uncompensable tasks, or composite tasks. Each composite task refers to a unique CWF_T-net at a lower level in the hierarchy.

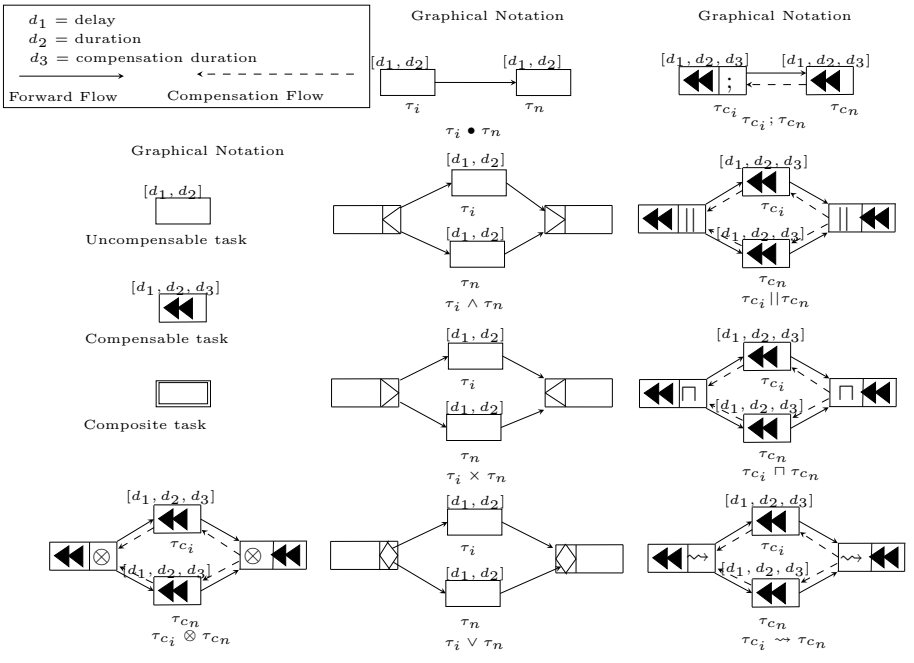


Fig. 2. Modeling Elements of CWML$_T$

Construction Principle: Construction principles for the graphical representation of tasks are as follows:

- The operators [•, ;] are used to compose the operand tasks sequentially. Atomic timed uncompensable tasks and atomic timed compensable tasks are connected by a single forward flow. Atomic timed compensable tasks are connected by a forward flow if they are composed using (•) and by both a forward flow and a compensation flow if they are composed using the sequential operator (;);
- A pair of split and join routing tasks are used for tasks composed by {$\wedge$, $\vee$, $\times$, $||$, $\sqcap$, $\otimes$, $\rightsquigarrow$}. Atomic timed uncompensable tasks are connected with split and join tasks by a single forward flow. Atomic timed compensable tasks are connected with split and join tasks by two flows (forward and compensation);
- For those operators that are associative, an n-fold composition (e.g., ($\tau_1 \wedge \tau_2$) $\wedge$ τ_3) is represented using the appropriate n-fold split and join (e.g., $\tau_1 \wedge \tau_2 \wedge \tau_3$).

CWML$_T$ is a block-structured language and hence the soundness is preserved by construction (soundness of a workflow requires that upon input of a token in the input condition, the underlying Petri net eventually terminates and at the moment it terminates, there is a token in the output condition and all other places are empty). The use of structured vs. unstructured workflow is debatable;

usually unstructured workflow languages are more expressive, but soundness is not preserved by construction. Among some popular workflow management systems, YAWL [12] uses workflow patterns for workflow modeling and its language is unstructured, while ADEPT2 [13] uses a block-structured language, and BPEL uses an unstructured language.

4 Verification of CWF$_T$-Nets

Once a workflow is modeled using a CWF$_T$-net, the underlying ETPN is translated into the DVE model specification by an automatic translator. Combining an LTL property with the DVE model specification, the DiVinE model checker determines whether or not the property holds.

A model described in DVE consists of processes, message channels and variables. Each process, identified by a unique name, consists of a list of local variable declarations, process state declarations, initial state declaration and a list of transitions. A transition may contain a guard (which decides whether the transition can be executed) and effects (which assign new values to local or global variables).

DiVinE is an un-timed model checker which generally cannot verify timed systems. Lamport [14] advocated *explicit-time description methods* using a general model construct, e.g., global integer variables or synchronization between processes commonly found in standard un-timed model checkers, to realize timed model checking. He presented an explicit-time description method using a clock-ticking process (*Tick*) to simulate the passage of time; we are using this method for our translation procedure.

Before we describe the translation procedure we define an ETPN representation in DVE, upon which we will perform the verification.

Definition 9. *A DVE$_T$ model is a 7-tuple, DM = (now, $Proc_{tick}$, Proc, PTimer, V, T'_P, S_0) where:*

- *"now" is a variable which indicates the current time,*
- *$Proc_{tick}$ is a clock process,*
- *$Proc = \{Proc_{tick}\} \cup \{Proc_1, Proc_2, ..., Proc_n\}$ is a finite set of processes,*
- *$PTimer = \{PTimer_1, PTimer_2, ..., PTimer_n\}$ is a finite set of variables, where $PTimer_i$ is the timer of $Proc_i$,*
- *$V \subseteq \{now\} \cup PTimer \cup \{v_1, v_2, ..., v_k\}$ is a finite set of variables,*
- *$T'_P = \{t'_{tick}, t'_1, ..., t'_m\}$ is a finite set of transitions, where each transition is associated with a set of guards, G, and a set of effects, E,*
- *$S_0 : V \to \{0, 1, 2, 3, ...\}$ is the initial assignment of the variables.*

Translation Principle: The underlying $ETPN$ model, $\mathrm{PM} = \langle P, T_P, F, M_0, m\rangle$ of a CWF$_T$-net is translated to a DVE$_T$ model DM = (*now*, $Proc_{tick}$, *Proc*, *PTimer*, *V*, T'_P, S_0) by the following rules:

i. for each place $p \in P$, there corresponds a variable v_p in DM; the initial values of the variables are set with the initial marking M_0 of PN, i.e., $\forall_{p \in P} S_0(v_p) = M_0(p)$;

ii. for each transition $t_i^s \in T_s$, which is associated with a set of pre-conditions $\{\psi_i^p\}$ and a set of actions $\{\psi_i^a\}$, there corresponds a process $Proc_i$ in DM; $Proc_i$ has a transition t_i' associated with a set of guards, G_i and a set of effects, E_i; G_i and E_i are determined by the pre-conditions, actions, time constraints and flow relations of t_i^s:
$G_i = \{\psi_i^p\} \cup \{S(v_p) \geq 1 \mid p \in {}^\bullet t_i^s\} \cup \{(now - PTimer_i) \geq delay(t_i^s)\}$;
$E_i = \{\psi_i^a\} \cup \{S(v_p) = 1 \mid p \in t_i^{s\bullet}\} \cup \{PTimer_k = now | (t_i^{s\bullet} \cap^\bullet t_k^s) \neq \emptyset$, where $t_k^s \in T_s\}$;

iii. for each transition $t_i^w \in T_w$, which is associated with a set of pre-conditions $\{\psi_i^p\}$ and a set of actions $\{\psi_i^a\}$, there corresponds a process $Proc_i$ in DM; $Proc_i$ has a transition t_i' associated with a set of guards, G_i and a set of effects, E_i; G_i and E_i are determined by the pre-conditions, actions, time constraints and flow relations of t_i^w:
$G_i = \{\psi_i^p\} \cup \{S(v_p) \geq 1 \mid p \in {}^\bullet t_i^w\} \cup \{(now - PTimer_i) \geq delay(t_i^w) \wedge (now - PTimer_i) \leq duration(t_i^w)\}$;
$E_i = \{\psi_i^a\} \cup \{S(v_p) = 1 \mid p \in t_i^{w\bullet}\} \cup \{PTimer_k = now | (t_i^{w\bullet} \cap^\bullet t_k^w) \neq \emptyset$, where $t_k^w \in T_w\}$;

iv. the *tick* process $Proc_{tick}$ works as the *Tick*, defined in Definition 8;

v. a transition t_i' in DM is *firable* if it satisfies its guard conditions. If t_i' is firable at state S, firable (t_i', S) is true, otherwise it is false; A *firable* transition may or may not fire depending on whether or not the event actually takes place.

A proof similar to the one in [5] will show that for any LTL formula ϕ, $PM \vDash \phi$ iff $DM \vDash \phi$.

5 Case Study and Property Verification

In this section, we model a health care workflow using CWF_T-nets and verify properties of the model. We study an Acute Abdominal Pain Diagnosis (AAPD) workflow, which we are developing with the local health authority, the Guysborough Antigonish Strait Health Authority (GASHA).

AAPD Workflow: GASHA has established a Seniors' Care program in response to an aging population and ongoing pressures on the health services delivery system created by the increasing number of individuals in hospital medical beds waiting for nursing home placement. The aim of the program is to create a seamless, integrated continuum of care for the senior population that improves accessibility to services across the district, provides timely intervention as needed and keeps service at the community level. We are building workflow models for each of the key sectors of GASHA's Seniors' Care program. A large number of seniors are admitted to the hospital emergency services with complaints of abdominal pain. The AAPD workflow for seniors overlaps both the hospital emergency department and the Seniors' Care program. We show how timed compensable workflow nets can help develop a verifiable health care workflow, in this case an AAPD workflow.

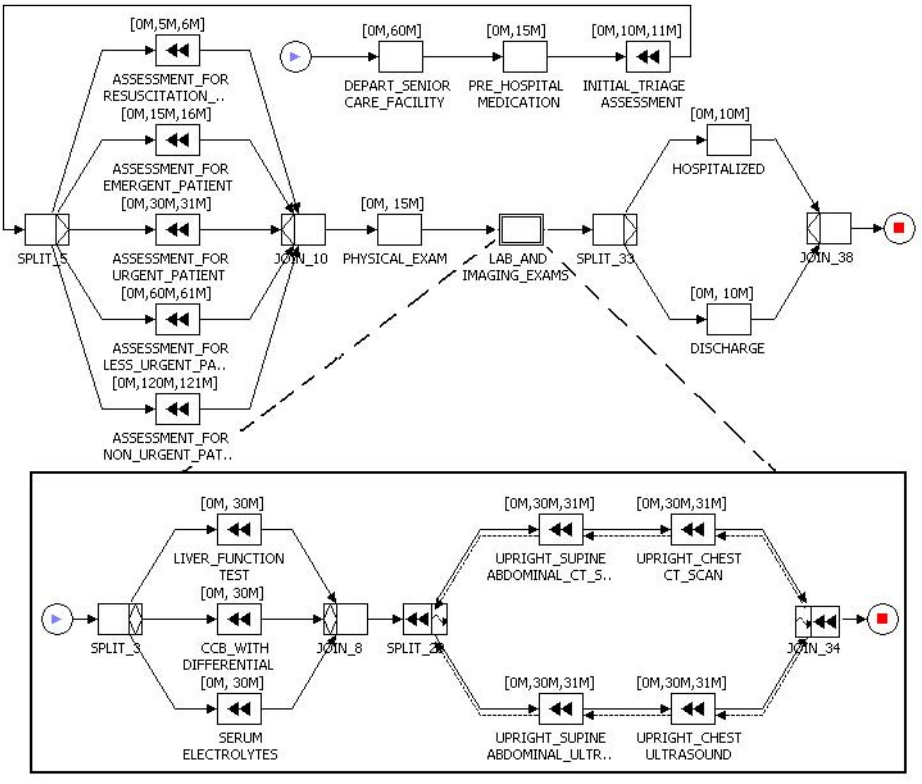


Fig. 3. An Acute Abdominal Pain Diagnosis Workflow for Seniors

In this workflow (see Fig. 3), designed using the graphical editor of *Nova WorkFlow*, a patient with acute abdominal pain is transferred to the hospital Emergency Department (ED) from a senior' nursing home or other location. According to the American Pain Society's Guidelines for the treatment of pain, each patient should receive individual optimal doses of pharmacological pain relief, which can be administered by the pre-hospital ambulance care unit. Proper assessment of an elderly patient after medication is required because of an increased risk of cerebral and cardiovascular negative effects when an opioid is administered [15]. After the patient has reached the ED, he is *Triaged* (emergency assessment to prioritize a patient's treatment) by a nurse. A patient should have an initial triage assessment within 10 minutes of arrival [16]. Thus we have assigned some *duration* to this task; if the task is not completed within its assigned *duration*, the compensation flow should be activated and raise some alert to handle the situation. According to the Canadian ED Triage and Acuity Scale, five Triage levels are defined: *Resuscitation*, *Emergent*, *Urgent*, *Less Urgent*, and *Non Urgent*. Depending on the *Triage* results, particular tasks need to be performed within a certain time frame (from immediate to 120 minutes). A Triage includes a primary survey *ABCD* (**A**irway and Cervical Spine,

Breathing, **C**irculation, and **D**isability (Neurological)) and a secondary survey *EFGH* (**E**xposure/**E**nvironmental Control, **F**ull set of vital signs, **G**ive comfort measures, and **H**istory and **H**ead to toe assessment). After Triage, the physician checks the patient's medical history, performs a physical exam and requests a series of lab exams (Complete Blood Count (CBC) with Differential and Liver Function Test) and imaging exams (CT scans for upright and supine abdominal, and upright chest) for the patient. The patient finishes the lab exams and waits for the CT scan films. The CT scan can fail for different reasons and raise an exception which can then be compensated (by a consultation with a physician) using an alternative flow, performing an Ultrasound for upright and supine abdominal, and upright chest. Finally the diagnosis workflow ends and the patient is either admitted to the hospital or discharged from the ED. Often, CT scan devices may not be available for a patient at a particular time as a hospital has a limited number of CT devices. The hospital may decide to use an Ultrasound instead of a CT scan when a CT scan is not available as a predefined compensation in the workflow. We model this scenario using an Alternative Forwarding ($S \rightsquigarrow T$) composition; here S (for CT scan) will try to execute first and if it fails then T (for Ultrasound) will execute as a compensation. The compensation may be done by different health services delivery professionals (e.g., nurse, administrative staff, physician).

The time information used in this model is obtained from the guidelines for nurses at GHASA. The entire Seniors' Care program model is large; hence the *state explosion* problem for the full model can be crucial. DiVinE exploits the power of distributed computing facilities so that more memory is available to accommodate the state space of the system model; parallel processing of the states can, moreover, reduce the verification time [17]. We are using the distributed LTL model checker DiVinE for these reasons. We are using a discrete clock to represent the passage of time, which might blow up the state space of the models where the time units of *delay*, *duration*, and *compensation duration* vary substantially. To address this issue we could incorporate the efficient EDM [18], in which the clock may *leap* multiple time units in a tick. This would greatly reduce the state space.

Property verification: Once we finish modeling the system, we need to identify the properties which must be true in the model. We verify the following properties of the model:

1. If the initial assessment of a patient is not made within 10 minutes of reaching the ED, the appropriate personnel should be alerted.
   G ((initial_triage_assessment_start ∧ initial_triage_assessment_timer10m ∧ ! initial_triage_assessment_completed) − > F compensated)
2. An emergent patient should be assessed within 15 minutes.
   G ((patient_assessment_result ∧ patient_emergent) − > F (assessment_emerge- nt_Patient ∧ assessment_emergent_patienttimer15m))
3. A patient should not be in the ED for more then 360 minutes.
   G (start − > F (end ∧ now_less_then360))

4. A problem with a CT device should be compensated within 30 minutes using an Ultrasound.
   G (problem_in_ct $->$ F (ultrasound_completed $\wedge$ ultrasound_timed30m))

We define *initial_triage_assessment_timer10m* as *ed_initial_triage_assessment-_timer <= 10*, where 10 means ten minutes; the other propositions in the LTL formula use variables available in the model. The experimental results show that properties 1 and 3 hold in the model and properties 2 and 4 do not hold. The counter example for property 2 shows that an emergent patient may not be assessed within the assigned time and in this situation in the model this task is compensated; the problem here is with property 2. The counter example for property 4 helps us find an execution sequence for which both the CT and Ultrasound do not occur in 30 minutes. In this case, both the execution sequences failed. To avoid the problem we should modify the model to include another path that will alert the appropriate personnel if both the execution sequences fail.

6 Related and Future Work

Health services delivery workflows are often complex and time sensitive. In order to ensure a patient's safety, health-related workflows should be verified and mechanisms for exception handling must be incorporated before they go into operation. Modeling a workflow with time constraints and compensation is not easy with existing tools. In this paper we provided a new language for modeling time-constrained workflows with compensation. We used t-calculus operators as they are expressive enough to represent many compensation scenarios. Our tool allows us to input a timed compensable workflow using a graphical editor and automatically translate the model into the input language of the DiVinE model checker; we can then verify LTL properties of the model. In our case study we used our tool to model and verify a Seniors' Care workflow. A timed compensation workflow model like the one we provide here is well suited for medical administrative workflows and laboratory workflows.

Over the years, researchers have investigated how to improve workflow systems to model real world scenarios and how to increase the reliability of a workflow. Acu et al. [19] extended the work on workflow nets by van der Aalst [12] using compensations; their approach was similar to the concept of compensation employed in [20]. Li et al. [4] defined the behavioral characteristics of the transaction calculus (t-calculus) operators focusing on compensable transactions. However, these formal models of compensation [4], [19], [20] lack the notion of time. In [21], the authors provided a formalism for timed workflow net based on Petri nets and gave an analysis of their boundedness and liveness properties. Medical errors (e.g., errors in a blood transfusion protocol and a chemotherapy process) were detected by the authors [22] using a model checking approach applied to medical processes defined by the Little-JIL graphical language. YAWL [12] comes with limited forms of verification (e.g., livelock, deadlock, etc.) and an ADEPT2 [13] workflow can be verified using SeaFlows [23]. The method in [21] and the tools in [22], [12], [13] do not have any compensation mechanism. BPEL has been used

in the industry for some time and there are many publicly available tools (e.g., WSEngineer [24]) to analyze a workflow designed in BPEL. While WSEngineer, which is based on CSP, can verify temporal properties in workflows, it cannot incorporate quantified time notion in its specification language. Timed automata and time Petri nets are two popular formalisms for timed model checking due to their simple graphical representations and sound mathematical formalisms. While the timed automata based model checker UPPAAL can be used to verify time-constrained compensable workflows, we favour the use of time Petri net formalisms for the same reasons provided in [25]. It is possible to design and verify a timed compensable workflow using time Petri net tools such as Romeo and TINA. However, the required modeling effort and the state explosion problem often limit the applicability of these tools for real world workflow models. Our graphical modeling language for timed compensable workflow significantly reduces the effort required to design a large timed compensable workflow system; use of DiVinE enables us to handle the huge memory requirement for complex real world models.

An interdisciplinary team in the StFX Centre for Logic and Information[2], consisting of researchers and students in computer science and health related fields, have been working closely with clinicians, administrators and other health care providers to greatly refine the details of the process of care and include information on time, access control and other process specific information. We are also developing a monitoring mechanism that can be included into the workflow model, and thus can facilitate the error handling during workflow execution. All these efforts will lead to the development of next generation workflow processes and information management systems which can enhance the potential of Canada's emerging electronic health record system and improve health services delivery by providing automated decision support.

Acknowledgment. This research is sponsored by NSERC, and by the Atlantic Canada Opportunities Agency. The computational facilities are provided by ACEnet. We also like to thank Dr. Hao Wang as well as Prof. Heather Jewers from the StFX School of Nursing and numerous clinicians from GASHA.

References

1. Rabbi, F., Wang, H., MacCaull, W.: Compensable workflow nets. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 122–137. Springer, Heidelberg (2010)
2. Mashiyat, A.S., Rabbi, F., Wang, H., MacCaull, W.: An automated translator for model checking time constrained workflow systems. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 99–114. Springer, Heidelberg (2010)
3. Han, M., Thiery, T., Song, X.: Managing exceptions in the medical workflow systems. In: The 28Th International Conference on Software Engineering (ICSE 2006), pp. 741–750. ACM Press, New York (2006)

[2] `http://logic.stfx.ca/`

4. Li, J., Zhu, H., He, J.: Specifying and verifying web transactions. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 149–168. Springer, Heidelberg (2008)
5. Rabbi, F.: Design, development and verification of a compensable workflow modeling language. M.Sc. Thesis, St. Francis Xavier University (expected 2011) Preliminary version, `http://logic.stfx.ca/~software/DDVCWML.pdf`
6. Li, W., Fan, Y.: A time management method in workflow management system. In: The 2009 Workshops at the Grid and Pervasive Computing Conference, pp. 3–10. IEEE Computer Society, Washington (2009)
7. Combi, C., Posenato, R.: Controllability in temporal conceptual workflow schemata. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 64–79. Springer, Heidelberg (2009)
8. Li, H., Yang, Y.: Verification of temporal constraints for concurrent workflows. In: Yu, J.X., Lin, X., Lu, H., Zhang, Y. (eds.) APWeb 2004. LNCS, vol. 3007, pp. 804–813. Springer, Heidelberg (2004)
9. van der Aalst, W.M.P., Van Hee, K.: Workflow management: models, methods and systems. The MIT press, Cambridge (2002)
10. Mashiyat, A.S.: Verification of time-constrained workflows in a distributed memory environment. M.Sc. Thesis, St. Francis Xavier University (May 2011)
11. Reynier, P.-A., Sangnier, A.: Weak Time Petri Nets Strike Back! In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 557–571. Springer, Heidelberg (2009)
12. van der Aalst, W.M.P., ter Hofstede, A.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
13. Reichert, M., Rinderle, S., Kreher, U., Acker, H., Lauer, M., Dadam, P.: ADEPT2 - next generation process management technology. In: Proceedings Fourth Heidelberg Innovation Forum, Aachen, D.punkt Verlag (2007)
14. Lamport, L.: Real-time model checking is really simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
15. Rantala, A., Ivarsson, K., Johansson, A.: Acute abdominal pain: pre-hospital evaluation of ketobemidone administration (Technical report)
16. Murray, M., Bullard, M., Grafstein, E., et al.: Revisions to the Canadian emergency department triage and acuity scale implementation guidelines (Technical report)
17. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core LTL model-checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
18. Wang, H., MacCaull, W.: An efficient explicit-time description method for timed model checking. In: 8th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 09). EPTCS, vol. 14, pp. 77–91 (2009)
19. Acu, B., Reisig, W.: Compensation in workflow nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 65–83. Springer, Heidelberg (2006)
20. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 209–220. ACM Press, New York (2005)
21. Tiplea, F.L., Macovei, G.I.: E-timed workflow nets. In: the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 423–429. IEEE Computer Society, Washington (2006)

22. Christov, S., Chen, B., Avrunin, G.S., Clarke, L.A., Osterweil, L.J., Brown, D., Cassells, L., Mertens, W.: Rigorously defining and analyzing medical processes: An experience report. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 118–131. Springer, Heidelberg (2008)
23. Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., Dadam, P.: On enabling data-aware compliance checking of business process models. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 332–346. Springer, Heidelberg (2010)
24. Foster, H., Uchitel, S., Magee, J., Kramer, J.: LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: The 28th International Conference on Software Engineering (ICSE) – Research Demonstration, pp. 771–774. ACM Press, New York (2006)
25. van der Aalst, W.: Three good reasons for using a Petri net-based workflow management system. In: The International Working Conference on Information and Process Integration in Enterprises (IPIC 1996), pp. 179–201 (1996)

Author Index

Ahmed, Ijaz 6
Aldini, Alessandro 165

Baier, Christel 228
Bel Hadj Amor, Zeineb 21
Belinfante, Axel 117
Bernardo, Marco 165
Brauer, Jörg 37
Büker, Matthias 149

Cataño, Néstor 6
Côté, Daniel 52

Damm, Werner 149
Daskaya, Ilyas 68
Dehning, Bernd 212
de Moura, Leonardo 5
Dierkes, Michael 102

Ehmen, Günter 149

Ferro, Luca 21
Fraikin, Benoît 52
Frappier, Marc 52

Ghafari, Naghmeh 212

Horauer, Martin 37
Huhn, Michaela 68

Joyce, Jeff 212

Katoen, Joost-Pieter 1
Klein, Joachim 228
Klüppelholz, Sascha 228

Kowalewski, Stefan 37
Kumar, Ramana 212

Lachaize, Jérôme 21
Lantreibecq, Etienne 180
Lefftz, Vincent 21

MacCaull, Wendy 244
Marinelli, Lawrence 117
Mashiyat, Ahmed Shah 244
Milius, Stefan 68
Mousavi, MohammadReza 134

Papailiopoulou, Virginia 85
Parissis, Ioannis 85
Pierre, Laurence 21

Rabbi, Fazle 244
Raffelsieper, Matthias 134
Rajan, Ajitha 85
Reinbacher, Thomas 37
Reniers, Michel A. 196

Serwe, Wendelin 180
Sijtema, Marten 117
Sproston, Jeremy 165
Stam, Andries 228
Stappers, Frank P.M. 196
St-Denis, Richard 52
Steininger, Andreas 37
Stierand, Ingo 149
Stoelinga, Mariëlle I.A. 117

Weber, Sven 196

Zamantzas, Christos 212